

Reducing the Power Consumption of IoT Devices in Task-Oriented Programming

Master's Thesis Computing Science

SJOERD CROOIJMANS

June, 2021

Supervisor:

dr. Pieter Koopman

Daily supervisor:

Mart Lubbers MSc

Second reader:

prof. dr. Sven-Bodo Scholz

Radboud University



Abstract

The Internet of Things (IoT) is a growing network of interconnected devices that interact with both humans and each other. IoT devices are often low-cost microcontrollers with limited resources. One of these resources is energy consumption. Improving the energy efficiency is better for the environment and increases the longevity of battery-powered devices. Hence, this thesis presents a method to reduce the energy consumption of devices running mTask programs.

The mTask language is a Task-Oriented Programming (TOP) language for microcontrollers. It is integrated with iTasks, a TOP language for distributed web applications. Together they allow the programmer to create IoT applications from a single source. TOP is a declarative programming paradigm to develop distributed and interactive systems. It focuses on what work needs to be done instead of how this work is performed.

A method is proposed to use the power modes of microcontrollers in mTask programs automatically and with minimal to no extra effort required from the programmer. It operates at a high abstraction level fitting the TOP paradigm. The proposed method reduces the energy consumption in two ways.

First, mTask programs do a lot of repetitive work, such as monitoring a sensor value. The former execution model of mTask performed this repetitive work as often as possible. Constantly evaluating tasks, even if no events are triggering the evaluation, is unwanted. For instance, the room temperature changes gradually, and it is a waste of energy to measure it at the fastest pace possible. The novel task execution model shown utilises dynamic evaluation frequencies. This model saves energy by using the sleep modes of microcontrollers in the time between task evaluations.

Secondly, the mTask language is extended with interrupts. Interrupts allow the device to monitor a sensor without continuously checking the state of this sensor. Furthermore, interrupts are superior over polling in detecting short-lived events.

Through analysis of several programs we show that the energy consumption of IoT devices running mTask programs is reduced significantly without requiring much effort from the programmer. The exact power reductions depend on the device and task, but our examples show an energy reduction of up to 88%.

Acknowledgements

The thesis I worked on for the past few months would not have been completed without the help of the following people: First, I would like to sincerely thank my supervisors Pieter Koopman and Mart Lubbers, who provided me with excellent feedback and taught me invaluable new skills. Moreover, I would like to thank them for our weekly meetings, where they kept me sharp by asking critical questions and helping me when I was stuck. Secondly, I want to thank Sven-Bodo Scholz for being the second reader of my thesis. I also would like to thank Daphne van Dijk, who helped me with all my questions regarding the English language and Colin de Roos for providing some feedback. Lastly, I would like to thank my friends, family and partner for their encouragements and support during the completion of this thesis.

Contents

1	Introduction	1
1.1	TOP for the IoT	2
1.2	Problem statement	2
1.3	Thesis outline	3
1.4	Notations and conventions	3
2	Background	5
2.1	Task-oriented programming	5
2.2	iTask	6
2.3	Energy-aware scheduling	7
3	mTask	9
3.1	The mTask language	10
3.1.1	Blinking an LED using mTask	11
3.2	Basic Tasks	12
3.3	Combinators	13
3.4	The mTask ecosystem	15
3.4.1	Server	15
3.4.2	Runtime system	16
4	Task scheduling in mTask	19
4.1	Evaluation interval	19
4.2	User-defined refresh rates	20
4.2.1	Data type for refresh rates	20
4.2.2	Adapting the refresh rate of sensor tasks	21
4.2.3	Adapting the refresh rate for receiving SDS values	22
4.2.4	Adapting the refresh rate of the repeat combinator	22
4.2.5	Alternative notations	23
4.2.6	Implementation	23
4.3	Deriving refresh rates	23
4.3.1	Implementation	26
4.3.2	Examples	26
5	Task scheduling on the device	27
5.1	Introducing the scheduling problem	27
5.1.1	Objectives	27
5.1.2	Concepts	28
5.1.3	Problem relaxations	30

5.2	Scheduling algorithm	31
5.2.1	Scheduling task evaluations	31
5.2.2	Scheduling power states	32
5.3	Properties of the scheduling algorithm	33
6	Interrupts	37
6.1	Interrupt types	38
6.2	Interrupts on the Arduino platform	38
6.3	Modelling interrupts	39
6.3.1	Examples	40
6.4	Handling interrupts	41
6.4.1	Registering and deregistering interrupts	41
6.4.2	Triggering interrupts	42
6.4.3	Evaluating interrupt tasks	42
7	Resulting power reductions	43
7.1	Methodology	43
7.2	Measurements	44
7.2.1	Blink	45
7.2.2	Thermometer	46
7.2.3	Light switch	46
7.2.4	Plant monitor & Thermometer	47
8	Conclusion	51
8.1	Conclusion	51
8.2	Related work	52
8.2.1	iTask	52
8.2.2	Functional Reactive Programming	52
8.2.3	Embedded operating systems	53
8.2.4	Other	53
8.3	Future work	54
8.3.1	Hardware	54
8.3.2	Scheduler	54
	References	55
	Acronyms	59
	Glossary	60
	Appendices	
A	Hardware Characteristics	61
B	Evaluation interval examples	66

Chapter 1

Introduction

The number of objects connected to the internet is growing rapidly. It is predicted that the number of connected objects reaches 83 billion by 2024 [1]. The networked interconnection of these objects is often referred to as the Internet of Things (IoT). The IoT results in a distributed network of devices that interact with both humans and other devices [2]. An IoT object is a broad concept that includes many types of computing devices, from low-cost microcontrollers to smartphones. IoT devices are used in all sorts of applications and in different sectors such as retail, agriculture, healthcare, and consumers goods. For instance, sensors in agriculture help to gain insights into the factors that influence crop production [3]. Another example that is more close to home is a smart fridge. A smart fridge is a fridge equipped with an internet connection to let users interact with it using a smartphone app. This app can, for example, be used to control the fridge's temperature or check its contents.

From an abstract perspective, this system consists of three tiers or layers: the fridge, a network layer, and the smartphone app. This three-layer model—as shown in Figure 1.1—is one of the more basic architectures proposed to model IoT applications. The perception layer of this model contains the objects and sensors that sense and interact with the environment. The network layer is responsible for the connection between devices and some processing of the exchanged data. Finally, the application layer delivers all the services of the application to the user [4].

The functionalities of these layers are often provided by different applications. For instance, the embedded devices in the perception layer run a different application than the app in the application layer. These applications then communicate through (custom) communication methods and protocols in the network layer.

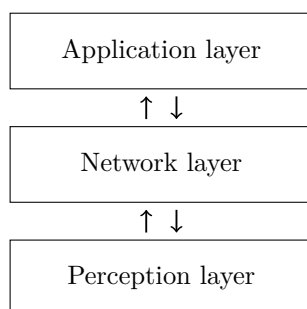


Figure 1.1: Three layer architecture of IoT applications.

1.1 TOP for the IoT

IoT applications are complex to develop due to the multiple components that also need to interact with each other. A tierless architecture solves this problem since all components and their interactions are generated from a single source. Creating tierless IoT applications is facilitated by the mTask [5] and iTask [6] language. These two Domain Specific Languages (DSLs) are based on the Task-Oriented Programming (TOP) paradigm. TOP is a novel programming paradigm to write multi-user, interactive and distributed systems in a declarative way [7]. The paradigm is, therefore, well suited for IoT applications which are also distributed and interactive. Moreover, the paradigm lets the programmer focus on the work to be done instead of how the work is done. The work to be done is captured in tasks. Tasks in TOP are similar to real-world tasks and represent a unit of work. For instance, a task could be showing a form to the user or monitoring the value of a sensor. In these tasks, details like the graphical representation of a form are hidden from the programmer. The ability to hide away implementation details makes TOP even more suited for the development of IoT applications. Details like the communication protocol and low-level details of the IoT devices are hidden so that the programmer can focus on the real logic of the application. The two DSLs both serve a different purpose in the IoT stack. The iTask language is suitable for specifying applications on general-purpose computing devices, whereas mTask is designed to specify programs for low-level and resource-restricted devices.

1.2 Problem statement

IoT devices are used for a broad range of applications and often have limited resources. Therefore, it is crucial to make the applications running on these devices as lightweight as possible. The focus of this thesis is to improve one of these resources, namely the power consumption. More precisely, improving the power consumption of programs written using the mTask language. Reducing the power consumption has multiple advantages. It improves the applicability and longevity of battery-powered devices. Moreover, it reduces the running costs and impact on the environment. The energy consumption is reduced in two ways.

Dynamic evaluation frequency A lot of work is repeated during the execution of mTask programs, like reading the value of a temperature sensor. This repeated work is in the current execution model of mTask performed continuously and as often as possible. However, in most cases, this is inefficient and unnecessary as the value (e.g. temperature) stays roughly the same in such a short time. The iTask language does not have this problem because it evaluates tasks based on events. Therefore, this thesis implements and proposes a novel event-driven execution model for mTask. This execution model performs the work based on clock-based events. Hence, work is only performed when a clock-based event triggers it instead of executing it at a continuous rate. This execution model is not only helpful in limiting the frequency of sensor readings, but it also prevents repetitive evaluations of a program that is, for instance, waiting for a delay to be over. The power consumption in this execution model is reduced by bringing the device to a low-power state in the obtained spare time.

Interrupts The polling rate of sensors is reduced to a more sensible rate using time-based events but, it is in a selection of cases even possible to avoid polling altogether using interrupts. Interrupts automatically ensure that the program is notified when a General Purpose Input/Output (GPIO) pin has a specific state. A selection of sensors can then trigger this GPIO pin. Thus, it is no longer necessary to poll the state of such a sensor,

which has two advantages. First, the power consumption is reduced since it is no longer necessary to wake up for reading sensor values. Secondly, events happening in between readings no longer remain unnoticed. Therefore, the execution model is extended to support interrupt-based events.

1.3 Thesis outline

The required background information of this thesis is presented in Chapter 2. It starts with a more in-depth explanation of TOP and iTask. Subsequently, background information about energy-efficient task scheduling is provided. Chapter 3 introduces the current state of the mTask language and implementation.

Chapters 4 to 6 introduce the extensions mTask. Chapter 4 presents a method to calculate the bounds on the next time a task should evaluate. The scheduler described in Chapter 5 overviews all tasks running on the device and selects the most optimal time within this interval to evaluate each task. The scheduler has, besides deriving a task schedule, another responsibility: selecting the most optimal sleep mode. Chapter 6 introduces a way to model interrupts in mTask. Chapter 7 discusses the results. Finally, Chapter 8 concludes and discusses this thesis and provides suggestions for future research.

1.4 Notations and conventions

The notation $[a, b]$ is used throughout this thesis to denote a timing interval containing all natural numbers between and including a and b . The time unit of this interval is milliseconds unless noted otherwise.

Furthermore, code segments are recognizable by a grey background. These code segments are decorated with a caption if it is a newly introduced definition. An example code segment is shown below.

Example code

The source code of mTask, including the adaptations made in this thesis, can be found at: <https://ftp.cs.ru.nl/Clean/mTask/2021-June-Sjoerd/>. Lastly, acronyms and definitions are explained in the Acronym list and Glossary.

Chapter 2

Background

This chapter discusses the background material of this thesis. The first two sections introduce TOP and iTask in more detail. Finally, the last section provides an overview of the background information concerning energy-aware scheduling.

2.1 Task-oriented programming

TOP [7] is a programming paradigm that specifies multi-user, interactive, and distributed systems in a declarative way. The paradigm focuses on the work that needs to be done and not so much on how it is done. It hides implementation details of, for instance, the GUI representation as much as possible. Instead, TOP allows the developer to abstractly describe the work to be done by the machine or user in the form of tasks.

Tasks Tasks are the core concept of TOP and describe a unit of work, similar to real work. Tasks can operate at a high level, such as monitoring a plant, or at a low level, such as reading a temperature sensor. The inner workings of these tasks are not important for the rest of the program, and only the effects are exposed as (intermediate) results. These results have one of the following three states:

- **NoValue** if the task has no observable value for other tasks.
- **Unstable** if the task has an intermediate observable value. This result is not fixed and could change in the future.
- **Stable** if the task has a final observable value. This value is emitted for all observations in the future.

Tasks without a value or an unstable value can transition to one of the other two states at any time. A task with a stable value can never transition to another state. This is illustrated by the state diagram in Figure 2.1.

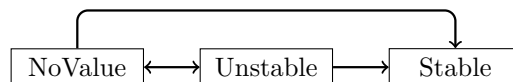


Figure 2.1: Transitions between task value states.

Tasks get a new value during evaluation. A task never finishes and can be evaluated multiple times or even indefinitely. No unnecessary work is done in a reevaluation because tasks reduce themselves to a new task containing all unfinished work during evaluation. Although it is valid to evaluate a task as many times as possible, it is unwanted as it would not always lead to progress. It is, therefore, more efficient to only evaluate a task when there is a reason to do so.

There is a distinction to be made between two types of tasks, namely basic tasks and task combinators. Basic tasks are the elementary units of the application. Combinators combine (basic) tasks into larger and more comprehensive tasks. These combinators combine tasks, for example, in parallel by evaluating them side by side or sequentially by continuing with another task if the first task has a specific value.

Shared Data Sources Task values are stored in memory and only observable by directly connected tasks. These two limitations are mitigated by storing data in a Shared Data Source (SDS). An SDS [8] is an uniform way to access data in arbitrary sources. SDSs hide implementation details of the underlying storage mechanism. Hence, it is possible to read and write from/to databases, text files, or memory in a uniform manner. SDSs also offer a way to communicate between tasks without passing the data directly through combinators. This is somewhat comparable to global variables. SDSs are flexible and can even be used to access read-only data sources such as the current time.

2.2 iTask

The iTask language [6, 9] is an Embedded Domain Specific Language (EDSL) to develop multi-user and distributed web applications using the TOP paradigm. The EDSL is hosted in the general-purpose functional programming language Clean [10]. The iTask system can be used to create both the front end and the back end of web applications. Most of it can even be generated automatically through generics. Generics are a type-independent way to specify a function. For instance, the Graphical User Interface (GUI) in the example below is automatically generated based on the type of `Plant`. The `enterPlant` task shows this form with the help of the `enterInformation` task. The fields of this form are based on the type `Plant`, a record of the species name, and a location. The species name is a string, and the location is an Algebraic Data Type (ADT) with four different constructors. The generic functions to show the form are made available by deriving the `iTask` class.


```
import iTasks

:: Location = LivingRoom | Kitchen | Garden | Other String
:: Plant = { species :: String, location :: Location }

derive class iTask Location, Plant

enterPlant :: Task Plant
enterPlant = enterInformation []

Start w = doTasks enterPlant w
```

Species: 


Location: 

Figure 2.2: GUI generated by the *enterPlant* task.

Event driven The execution model of the iTask system is already energy efficient since it only evaluates tasks when there are relevant events. The system waits the remaining time and it is up to the operating system to use this time efficiently. The event-driven execution model is created by implementing tasks as a state transforming function with as input an event and the current state. An event could, for example, be a new value in a text field or a refresh of the web page. The function then returns a task value, changes to the UI, a task continuation, and a new state. The task value contains the current value of the task and its stability (e.g. NoValue, Unstable or Stable). The task continuation is a new task containing all the unfinished work of the old task. The resulting simplified type of a task in the iTask system is:

$$Task : Event State \rightarrow (TaskValue, UiChange, Task, State)$$

Tasks require an engine to work correctly. This engine is included in the iTask system and provides an environment to run the tasks. This includes creating events, providing browser-side functionality, and applying the UI updates.

2.3 Energy-aware scheduling

By scheduling time-triggered task evaluations efficiently, the energy consumption of the device running these tasks can be improved. Such a scheduler closely resembles other real-time energy-aware schedulers in, for instance, operating systems. This section discusses the energy-aware scheduling techniques in literature that served as inspiration for the scheduling algorithm in mTask. There are many energy-aware scheduling techniques proposed in literature based on multiple techniques [11, 12].

This thesis focuses on Dynamic Power Management (DPM) to reduce power consumption. DPM reduces the power consumption by switching the device to a low power state when the CPU has no work to do. The device consumes less power in a sleeping state, but it can, in most cases, no longer execute jobs. Moreover, there is an overhead for switching between power states. DPM algorithms schedule jobs while taking the power states of the device into account.

A simple and in practice used approach is to finish the waiting jobs as fast as possible and sleep until a new task arrives. This so-called race-to-sleep principle is, for example, used in FreeRTOS [13]. A more sophisticated approach is introduced by Lee et al. [14]. They propose an algorithm called Leakage-Control EDF (LC-EDF), which is an extension of the Earliest Deadline First (EDF) scheduling algorithm. An EDF scheduler executes jobs in the order of their deadline. Jobs with an earlier deadline are prioritized over jobs with a later deadline. LC-EDF is a variant on EDF scheduling that delays the execution of jobs as long as possible and uses the freed-up time to sleep. Another algorithm that influenced the algorithm used in mTask is Enhanced Race-To-Halt (ERTH) [15]. ERTH is a partially on- and offline scheduling algorithm. The most relevant part of this algorithm is—in the light of this thesis—the mechanism to select the power state. ERTH uses the break-even time to determine the power states. The break-even time

is defined as the amount of time the device must be without work before switching to that power state becomes beneficial. All power states, relevant according to the break-even time, are then compared with a formula to calculate the power consumption. The power states and other characteristics of devices with an mTask implementation are discussed in Appendix A.

Chapter 3

mTask

IoT applications traditionally have a layered architecture. Each layer contains a part of the functionality, such as collecting or representing data. These functionalities are conventionally provided by separate applications written in different programming languages. For instance, the device collecting the sensor data runs a different program than the webserver presenting this data to the user. This diverse collection of programs makes the development of IoT systems complex. The mTask language [5] in combination with iTasks solves this problem. The mTask language is a TOP language to program low-cost and resource-restricted devices. It forms together with iTask a tierless IoT stack [16]. In this stack, the programmer develops all components and interactions of an IoT system as a single program.

This program is a Clean program in which mTask and iTask are implemented as two EDSLs. An EDSL is a language designed for a specific domain embedded within a host language. This host language is in the case of mTask and iTask, the functional programming language Clean. The domain of the mTask language is specifying programs for low-cost and resource-restricted devices, whereas iTasks is designed to specify multi-user and distributed web applications. These two languages are closely integrated. It is, for instance, possible to share data between the two TOP implementations using SDSs and run mTask as if they were regular iTask tasks.

An essential difference between mTask and iTask are the hardware requirements of the devices executing the programs. The requirements to run mTask programs are minimal as it can run on devices with as low as 2 KiB of memory and a slow 16 MHz processor. The iTask system, and consequently Clean, cannot run on these resource-restricted devices since it would require more memory, a faster processor, and an operating system. For this reason, mTask is developed alongside iTask to program microcontrollers with limited resources.

The mTask system has low requirements as it not evaluates the tasks in the host language. Instead, the IoT application running on the server transforms the task at runtime into byte code. Subsequently, the byte code is sent to one of the devices. The Runtime system (RTS), running on these devices, interprets the byte code and evaluates the task. This results in the architecture shown in Figure 3.1.

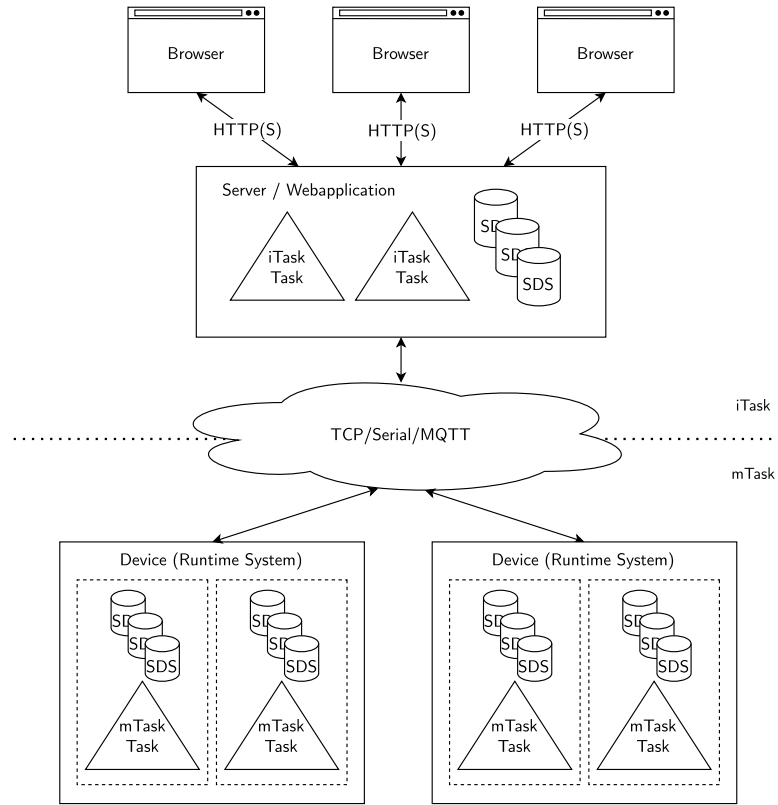


Figure 3.1: Architecture of IoT applications developed using mTask and iTask.

3.1 The mTask language

This section provides an overview of the mTask language. The mTask language is embedded within its host language using classes. This technique is also called tagless shallow embedding [17]. This means that the language constructs of the DSL are represented as overloaded functions. More concretely, the definition of the `Not` function is implemented like this:

```
class expr v where
  Not :: (v Bool) -> v Bool
  ...
```

The `Not` function is part of the `expr` class. By implementing instances of this class are views of the DSL created. A view is an interpretation of the DSL, such as a byte code representation or a pretty printer. The type of a view is represented by the type variable `v`. In the example, the `Not` function needs a `Bool` in view `v` as input and returns a `Bool` in that same view.

An example implementation for evaluator view is:

```
instance expr Eval where
  Not (Eval x) = Eval (not x)
  ...
```

This instance implements the `rtrn` class for the type `Eval`. `Eval` is a type to represent the evaluator view. It is a phantom type as it is only used to ensure type safety.

3.1.1 Blinking an LED using `mTask`

The `mTask` language is introduced by an often-used example for microcontrollers: blinking an LED. Blinking is the “Hello world!” of the microcontrollers due to the lack of a screen for showing text. The blinking example is implemented as a task and, later on, sent to the device for execution. The definition of this task and corresponding line-by-line explanation is given below.

```
1 blink :: Main (MTask v Bool) | mtask v
2 blink
3   = declarePin D13 PMOutput \led->
4     fun \blink=(\x->
5       delay (lit 1000)
6         >>|. writeD led x
7         >>=. blink o Not)
8     In {main=blink (lit True)}
```

Line 1: Type signature of `blink`. Decomposed from the outside to the inside the first encountered type is `Main`. `Main` is a record to differentiate the top level of an `mTask` program from a lower level in the type system. It has one field `main` to which the task is assigned. The type `Main` has one type argument for the type of this task. The type signature of an `mTask` task is `MTask v t`, where `v` is the view and `t` the type of the task value. `Blink` is thus an `mTask` task of type `Bool` with a still overloaded view. The view has a class constraint `mtask`. This is the collection of classes containing all the language constructs of `mTask`.

Line 3: Declares pin D13 and sets the pin mode to output. The variable `led` refers from now on to pin D13. The onboard LED of the Adafruit Feather is attached to this pin.

Line 4: Declares the function `blink`. This function has one argument, namely `x`. This variable is of type `Bool` and indicates whether the LED should be turned on or off.

Line 5: The `delay` task is a basic task to wait for a certain amount of time. It is unstable by default but becomes stable after 1000 ms.

Line 6: The delay of the previous line is combined with the `writeD` task using the `>>|.` combinator. This combinator waits until the `delay` task becomes stable and continues with the `writeD` task afterwards. In other words, the value of `x` is written to pin D13 after the delay is over.

Line 7: Recursively calls the `blink` function, but with the negation of argument `x`. The current value of `x` is emitted by `writeD` task, which emits the written value a task value. The `writeD` task and function call are connected through the `>>=.` combinator. This combinator steps when

the left-hand side is stable and passes the value of the task on the left-hand side to the function on the right-hand side. Hence, the written value is passed to the `Not` and `blink` function.

Line 8: Defines the starting point of the task. It is in this case the `blink` function with the argument `True`.

3.2 Basic Tasks

Basic tasks are the elementary units of TOP programs. Basic tasks do, on the contrary to combinators, not require other tasks to operate. This section gives an overview of the basic tasks available in `mTask`.

Peripherals Multiple types of peripherals like temperature, humidity, and light sensors are implemented in `mTask`. Interaction with these peripherals happens through one or multiple basic tasks. A selection of the functions to create these tasks is shown below. Functions to create other peripheral tasks are similar and omitted for brevity.

```
class dht v where
  temperature :: (v DHT) -> MTask v Real
  humidity    :: (v DHT) -> MTask v Real
  ...

class dio p v | pin p where
  writeD :: (v p) (v Bool) -> MTask v Bool
  readD  :: (v p) -> MTask v Bool | pin p
```

Reading sensor values works in a more or less similar way for each sensor type. The tasks are created by a function that takes the sensor or pin as input and returns the task. The returned task constantly emits the sensor reading as an unstable value. Take the `readD` task with one argument of type `v p`, where `p` must instantiate `pin`. In other words, this argument is a pin in the view `v`. It is thus an argument in the EDSL instead of the host language. The small example below reads the value of pin D13 and emits it as an unstable value.

```
declarePin D13 PMInput \d13 -> {main = readD d13}
```

Oppositely to the sensor tasks, there are also tasks to output to the environment, such as `wroteD`. The `wroteD` task writes a value to one of the GPIO pins. The written value is subsequently emitted as a stable value by the task. The example below shows a task to write the value high to pin D13.

```
declarePin D13 PMOutput \d13 -> {main = wroteD d13 (lit True)}
```

SDS SDSs in `mTask` offer a uniform way to store and receive data. The data in an `mTask` SDS is stored in local memory. There are three functions defined to create tasks for manipulating SDSs:

```
class sds v where
  setSds  :: (v (Sds t)) (v t) -> MTask v t | type t
  updSds  :: (v (Sds t)) ((v t) -> v t) -> MTask v t | type t
  getSds  :: (v (Sds t)) -> MTask v t | type t
  ...
```

Assigning new values to SDSs is done using the `setSDS` or `updSDS` functions. Both functions return a task to store a new value in the SDS, but the latter also atomically updates the value based on the old value. The value of the SDS is read with the task returned by the `getSDS` function.

It is also possible to convert an `iTask` SDS to an `mTask` SDS. The example below demonstrates this by incrementing the value of `mSDS`—an SDS shared by `iTask`—with one. After the increment, the new value is sent to the server to keep the `iTask` SDS in sync.

```
liftsds \mSDS=iSDS In {main = updSds mSDS \x -> x +. lit 1}
```

Other Three other relevant but not yet discussed basic tasks are the `rtrn`, `unstable`, and `delay` tasks.

```
class rtrn    v :: (v t) -> MTask v t | type t
class unstable v :: (v t) -> MTask v t | type t
class delay  v :: (v n) -> MTask v Long | type n & long v n
```

The `rtrn` and `unstable` task emit the value provided as an argument as a stable and unstable value, respectively. The `delay` task emits the time left as an unstable value until the delay is over and the time it overshot as a stable value afterwards.

3.3 Combinators

Combinators take one or multiple tasks as input and combine them into new tasks. There are three types of combinators available in `mTask`: `sequential`, `repeat` and `parallel`.

Sequential The `sequential` combinator combines tasks by evaluating them after each other. Tasks on the left-hand side of `sequential` combinators are continuously evaluated until a particular condition is reached. The task on the left-hand side is then removed and replaced with one of the continuations on the right-hand side. An overview of all `sequential` combinators is shown on the next page.

```

class step v | expr v where
  (>>=.) infixl 0 :: (MTask v t) ((v t) -> MTask v u) -> MTask v u | ...
  (>>|. ) infixl 0 :: (MTask v t) (MTask v u) -> MTask v u | ...
  (>>~.) infixl 0 :: (MTask v t) ((v t) -> MTask v u) -> MTask v u | ...
  (>>..) infixl 0 :: (MTask v t) (MTask v u) -> MTask v u | ...
  (>>*. ) infixl 1 :: (MTask v t) [Step v t u] -> MTask v u | ...

:: Step v t u
= IfValue      ((v t) -> v Bool) ((v t) -> MTask v u)
| IfStable     ((v t) -> v Bool) ((v t) -> MTask v u)
| IfUnstable   ((v t) -> v Bool) ((v t) -> MTask v u)
| IfNoValue    (MTask v u)
| Always       (MTask v u)

```

The `>>*` combinator is the most versatile combinator with support for complex step conditions. These conditions and corresponding continuations are combined into a list on the right-hand side of the combinator. This list of conditions is during evaluation matched against the value of the left-hand side until a match is found. All other combinators can be implemented using this combinator.

The four other combinators are shorthands for frequently used step conditions. The `>>=`, and `>>~`, pass the value of the left-hand side task to the function on the right-hand side and continue with the result. Analogous to the `bind` monad to combine two monads. The difference between the two combinators is that the `>>=` combinator requires the task value of the task on the left-hand side to be stable. Contrary to the `>>~`, that only requires a value on the left-hand side. The other sequential combinators behave similarly but do not pass the result of the task to the right-hand side. The example below repeatedly reads an analog pin until the value is smaller or equal than 10.

```

declarePin A1 PMInput \p ->
  {main = readA p >>*. [IfValue (\x. x <= (lit 10)) rtrn]}

```

Repeat The `rrepeat` combinator makes it possible to evaluate a task repeatedly by restarting a task when it becomes stable. The `rrepeat` function is defined as follows:

```

class rrepeat :: (MTask v a) -> MTask v a | type a

```

The `rrepeat` function needs a task and returns a new task to evaluate the provided task repeatedly. The value of the inner task is emitted as an unstable value if it has a value and no value otherwise. The example below continuously reads the value of pin A1 and writes it to A2.

```

declarePin A1 PMInput \a1 ->
declarePin A2 PMOutput \a2 ->
  {main = rrepeat (readA a1 >>~. writeA a2)}

```

Parallel Parallel combinators combine two tasks and evaluate them in parallel to each other. There are two parallel combinators available:

```
class (.&&.) infixr 4 v :: (MTask v a) (MTask v b) -> MTask v (a, b) | ...
class (.||.) infixr 3 v :: (MTask v a) (MTask v a) -> MTask v a | ...
```

The `.&&.` combinator emits the value of both tasks. The value of this task is stable if both tasks are stable and unstable otherwise. No value is emitted if at least one of the tasks has no value. Secondly, the `.||.` combinator emits the value of one task. This is the value of the most stable tasks and the value of the left-hand side if both tasks are equally stable. This combinator also has no value if none of the tasks has a value.

The example task below monitors two digital pins simultaneously using the parallel combinator and becomes stable as soon as one of the pins becomes high.

```
declarePin D10 PMInput \d10 ->
declarePin D11 PMInput \d11 ->
  let monitor pin = readD pin >>*. [IfValue id rtrn]
  in {main = monitor d10 .||. monitor d11}
```

3.4 The mTask ecosystem

The mTask language in itself is not sufficient for developing IoT applications. It also requires a way to translate the program into byte code, methods to communicate with the device, an RTS to interpreted the byte code and more. This functionality is all included in the mTask ecosystem. The mTask Clean library provides the server-side functionality, and the RTS contains the functionality for the device.

3.4.1 Server

The mTask library contains the embedded DSL and its views but also helper functions for sending the tasks to devices. There are currently three different views available.

- **Pretty Printer** The pretty-printer view transforms the task into a user-readable and well-formatted representation of the source code.
- **Simulator** The simulator view allows the user to interactively simulate tasks in iTask without the RTS.
- **Byte code** The byte code view generates byte code for interpretation on devices. This view compiles the EDSL on the server and transforms it into mTask specific byte code.

The byte code view is the most relevant view for this thesis and will be examined in more detail using an example.

Compiling and transmitting the previously explained blink task to a device with hostname “dev1” is demonstrated below.

```
main :: Task Bool
main = withDevice {TCPSettings | host="dev1", port=8123, pingTimeout=?None}
      \dev -> liftmTask blink dev
```

A connection is created between the server and the device before compiling the task. The `withDevice` function creates this connection. This function expects a communication method of which there are three available: TCP, Serial, and MQTT.

Transmission Control Protocol (TCP) is a communication protocol for communicating within networks. One of its use cases is sending tasks wirelessly to devices with WiFi capabilities such as the Wemos D1 mini. The serial communication method is used to transfer tasks over a serial connection. Lastly, Message Queuing Telemetry Transport (MQTT) is a protocol that usually operates on top of a TCP stack to make the connection more resilient for offline and sleeping devices. All these communications methods are implemented as an instance of the `channelSync` class.

The running example passes a `TCPSettings` record—with the settings for a TCP connection—to the `withDevice` function. This function uses this information to connect to the device and produces a device handle for the function provided as a second argument.

This device handle is used by the `liftmTask` for sending the compiled task to the device. This function also creates a bridge between `mTask` and `iTask` by constantly emitting the value of the `mTask` task in the `iTask` environment. This complete process is called *lifting* a task from `mTask` to `iTask`. The mechanics behind compiling a task becomes more clear by inspecting the type signature of the `liftmTask` function. The first argument of this function is the task to compile. However, the task type differs from the type signature of `blink` since the previously undefined view is replaced with `BCInterpret`. `BCInterpret` transforms the task into byte code.

```
class channelSync a :: a (sds () Channels Channels) -> Task () | ...
withDevice :: a (MTDevice -> Task b) -> Task b | ...
liftmTask :: (Main (MTask BCInterpret u)) MTDevice -> Task u | ...
```

3.4.2 Runtime system

The RTS is the part of `mTask` responsible for interpreting the byte code generated by the DSL. This RTS is running on the device and provides a foundation to run tasks on the fly. The system can be adapted to run on almost any device and is currently available for personal computers and a selection of Arduino-supported devices.

The responsibilities and mechanics behind the RTS are explained in more detail using the `mTask` event loop. The `mTask` event loop continuously executes the following phases:

Communication The server and device constantly exchange messages with each other. For example, the server sends new tasks to the device, and the device sends the values of these tasks back to the server. All communication received by the device is processed in this communication phase. New tasks received in this phase are saved in memory and prepared for execution in the next phase.

Evaluation In this phase, the tasks are evaluated on the device using a rewrite engine and an interpreter [18]. The interpreter creates, during the first evaluation, an initial task tree. A task tree is a hierarchical tree structure to represent the current state of a task. Basic tasks are the leaves of this tree, and all other nodes are combinators used to combine the subtrees into the final task tree. For example, Figure 3.2 shows the task tree of the `blink` task. A task tree is, under most circumstances, not completely unfolded in memory as this would lead to inefficiencies and an infinitely large tree in the case of recursion. Instead, the task tree on the right-hand side of a sequential combinator is often not unfolded until the step condition is met.

During an evaluation, the task tree of a task is in small steps rewritten by the rewrite engine. The rewrite engine walks through the tree and rewrites each node. Rewriting a task node can involve many different activities. For example, a node of the digital read task reads the value of a GPIO pin and emits the value as a task result. Another example is the step combinator (`>>*. .`), that first rewrites the left-hand side and then uses the interpreter to find a matching continuation. What is the same between all nodes is that rewrite steps are kept small. This makes multitasking in mTask easy because tasks are interleaved naturally due to the small rewrite steps [19].

Tasks in mTask are evaluated every loop iteration of the event loop. The event loop in mTask runs continuously with a maximum delay of 5ms between iterations. This is different from iTask, which executes the event loop and evaluates tasks based on events.

Garbage collection The mTask RTS often runs on low-cost devices with limited memory. Therefore, the RTS has its own memory management system to use the available memory efficiently. This memory management system allocates a large block of memory during the RTS' startup. In the garbage collection phase, all trashed items are removed from this memory block to create space for newly allocated items in the subsequent cycles of the event loop.

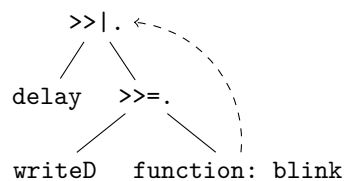


Figure 3.2: Task tree of the `blink` task.

Chapter 4

Task scheduling in mTask

The execution model of mTask is adapted to reduce the energy consumed by the device. The aim of this novel execution model is not to slow the system down but to sleep whenever it is unnecessary to evaluate tasks. The event loop of the former execution model iterated at a continuous rate and evaluated all tasks. This model is altered in two places to improve efficiency. The delay between loop iterations is made variable and, only the relevant tasks are evaluated instead of all tasks. These two changes improve the energy efficiency since the device can sleep in the time between loop iterations. This chapter explains how the variable time between iterations is derived at the task level. Every task has an interval to indicate when the next iteration should take place. This interval is called the evaluation interval of a task. The next evaluation time is expressed as an interval instead of a single point in time to archive better energy savings when there are multiple tasks. The details behind this are explained in Chapter 5.

4.1 Evaluation interval

The evaluation interval of a task is calculated at runtime by traversing the task tree of the task. New and therefore unevaluated nodes¹ of a task tree are evaluated immediately. Hence, the evaluation interval of a task with unevaluated nodes is equal to $[0,0]$. The importance of directly evaluating tasks becomes clear when inspecting the blink task again:

```
blink :: Main (MTask v Bool) | mtask v
blink
  = declarePin BuiltinLEDPin PMOutput \led->
    fun \blink=(\x->
      delay (lit 1000)
      >>|. writeD led x
      >>=. blink o Not)
    In {main=blink (lit True)}
```

It is unwanted that the device sleeps before the first evaluation of the `delay` task, write to the GPIO pin and the recursive call to the `blink` function. Since the goal of the new execution

¹Only the task tree nodes set to evaluate in the next evaluation step determine whether a task tree is unevaluated. For instance, the `>>|. operator` has nodes on the right-hand side. These nodes are never evaluated in the next evaluation step and thus ignored.

model is not to slow the system down but to sleep whenever it is unnecessary to evaluate a task. For instance, extra evaluations of the `delay` task before the delay is over are unnecessary. Thus, it is safe to postpone the next evaluation to a later time. The bounds on the time to the next task evaluation is called the evaluation interval. This evaluation interval is based on the refresh rate. The refresh rate differs from the evaluation interval as the refresh rate is defined for every task node in the task tree, whereas the evaluation interval is defined for a complete task i.e., a complete task tree. The evaluation interval is equal to the refresh rate of the task tree's root if the task tree contains no unevaluated nodes. If the task tree contains unevaluated nodes the evaluation interval is equal to `[0, 0]`. The value of the refresh rate is based on the task type, task state and child task nodes in the case of a combinator. The value is not fixed and can change after each evaluation. It is automatically derived based on the node's properties, but it is also desirable to make the refresh rate of some node types user definable.

4.2 User-defined refresh rates

Not all refresh rates are as set in stone as the refresh rate of the delay task. It is unclear and dependent on the use case what a reasonable refresh rate for some task type is. Take the task nodes that constantly emit the value of a sensor. The value of the sensor may constantly fluctuate, but it might not be necessary to evaluate the task node constantly. Evaluating the task node every 5 minutes might be sufficient, depending on the use case. Hence, this information cannot be derived automatically. This problem is solved by deriving a default refresh rate that the user can refine. A refinable refresh rate is implemented for all tasks receiving a sensor or SDS value and the `rrepeat` combinator. The required changes to the language are explained after introducing the data type for refresh rates.

4.2.1 Data type for refresh rates

The adaptations to the DSL rely on an ADT for timing intervals. The definition of this type is shown in Listing 4.1. The `TimingInterval` ADT has four different constructors, of which three have two variants. The first constructor is the `Default` constructor, which defines the default refresh rate. This `Default` constructor is necessary to avoid patching of unchanged refresh rates at runtime. Without this constructor, the refresh rate must always be contained in the byte code, but that is a waste of space if it is unchanged. The three other constructors are the `Before*`, `Exact*` and `Range*` constructors. The `BeforeSec` and `BeforeMs` constructors denote a refresh rate with just an upper bound. The tasks with a refresh rate created by these two constructors want to be evaluated before the specified time relative to the last evaluation. This implicitly means that the lower bound is set to zero. The `ExactSec` and `ExactMs` constructors denote an interval with only one value. Tasks with a refresh rate created by these constructors want to be evaluated at exactly that point in time after the last evaluation. The last constructors, `RangeSec` and `RangeMs`, denote an upper and lower bound on the next evaluation time.

The `Before*`, `Exact*` and `Range*` constructors have two variants, one for an interval in milliseconds and another for seconds. In this way, it is possible to represent a large number of intervals using the default 16-bit integer. It is important to notice that the scheduler tries to postpone task evaluations as long as possible. The lower bound on the refresh rates of the `Exact*` and `Range*` constructors is thus merely an additional guarantee to prevent extra evaluations of time-critical or resource-intensive task nodes.

```

:: TimingInterval v
= Default
| BeforeMs (v Int)           // equals [0, x]
| BeforeSec (v Int)          // equals [0, x * 1000]
| ExactMs (v Int)            // equals [x, x]
| ExactSec (v Int)           // equals [x * 1000, x * 1000]
| RangeMs (v Int) (v Int)    // equals [x, y]
| RangeSec (v Int) (v Int)   // equals [x * 1000, y * 1000]

```

Listing 4.1: ADT for timing intervals.

The `TimingInterval` ADT has one type argument `v`. This is the view used to raise the constructor arguments to the DSL. The values of these arguments may therefore be determined at runtime and on the device. As a result, it is possible to change the values supplied to the constructor dynamically, but not the kind of constructor. Another approach that allows for dynamic constructor kinds is to raise the complete constructor to the DSL. For instance, take this artificial function to create a sensor task:

```

sensor :: (v TimingInterval) (v Sensor) -> MTask v Int

```

This approach works but requires extra DSL constructs for creating timing intervals at runtime and it increases the bytecode's size. Therefore, only raising the constructor arguments is preferred over raising the complete constructor.

4.2.2 Adapting the refresh rate of sensor tasks

Sensor values do not change at fixed points in time. Hence, monitoring a sensor often means polling the sensor continuously for new values. The desired refresh rate for receiving new values is different per use case and should therefore be adaptable by the user.

New functions are introduced to create tasks with customized refresh rates. These new functions exist alongside the original functions to keep the language backwards compatible and keep programs with an unchanged refresh rate simple. The original functions are implemented as a macro of the new functions but with the default refresh rate already supplied. An overview of the changes to the `dht` and `dio` classes is shown in Listing 4.2. The adaptations to other classes are similar and omitted for brevity.

```

class dht v where
  temperature` :: (TimingInterval v) (v DHT) -> MTask v Real
  temperature  :: (v DHT) -> MTask v Real
  humidity`    :: (TimingInterval v) (v DHT) -> MTask v Real
  humidity     :: (v DHT) -> MTask v Real

class dio p v | pin p where
  readD` :: (TimingInterval v) (v p) -> MTask v Bool | pin p
  readD  :: (v p) -> MTask v Bool | pin p

```

Listing 4.2: Overview of the new function definitions for the DHT sensor and digital read. Only the new and original tasks are shown, other functions are left out for clarity.

4.2.3 Adapting the refresh rate for receiving SDS values

SDSs shared by `iTask` receive updates from the server when this SDS changes in `iTask`. These new values are not received during sleep for two reasons. New values of SDSs are processed at the start of the event loop, and the connection with the server is sometimes closed. Due to this reason, the `getSDS` task needs an adjustable refresh rate. The definition of the new `getSDS`` function to create the `getSDS` task with a customizable refresh rate is shown in Listing 4.3.

```
class sds v where
  getSDs` :: (TimingInterval v) (v (Sds t)) -> MTask v t | type t
  getSDs  :: (v (Sds t)) -> MTask v t | type t
```

Listing 4.3: Task variant for receiving SDS values with a user-definable refresh rate.

4.2.4 Adapting the refresh rate of the `rrepeat` combinator

The `rrepeat` task restarts the child task in the next evaluation if it is stable. However, it is sometimes desirable to postpone the restart of the child. The `rrepeatEvery` task is therefore introduced. The `repeatEvery` task is a variant of the `rrepeat` task with an adjustable refresh rate. A `rrepeatEvery` task with a non-zero interval does not reset the inner task immediately when it becomes stable. It instead waits until the lower bound of the refresh rate is over before restarting the child task. The difference between the `repeatEvery` and `rrepeat` task is best illustrated by two equivalences in pseudocode²:

```
rrepeatEvery rr body
  ≡ body .&&. delay (lowerBound rr) >>|. rrepeatEvery rr body
rrepeat body
  ≡ body >>|. rrepeat body
```

The `rrepeatEvery` task waits until the body is stable and the delay is over before restarting the task. Contrary to the `rrepeat` task, which restarts itself in the evaluation after the body becomes stable. This also shows the difference between using a `repeatEvery` task to postpone the task restart and inserting an `delay` task at the end of the loop body. The `repeatEvery` task starts counting the time until the next restart when the body restarted or when the restart deadline passed if the body became stable after the restart deadline. Intuitively, the loop body never restarts before it becomes stable or the restart delay is over. The restart could thus be later than the interval given to the `rrepeatEvery` task if the evaluation took longer than the restart time. This could lead to time drift. As a solution, the next restart deadline is moved forward by the time the restart was overdue. The interval to perform the loop body is thus independent of the evaluation time. Contrary to the `delay` task, which starts the wait time in the first evaluation of the `delay` task. The `rrepeat` task with `delay` task as a child causes a time drift due to the execution time of the rest of the loop body. Listing 4.4 shows the function definition for creating the `repeatEvery` task.

²The `repeatEvery` task is simplified. In practice, the delay starts counting when the is body restarted if it was not overdue. Otherwise, the restart time starts counting when the previous iteration became overdue. In other words, when the upper bound on the restart time of the previous iteration passed.

```
class rpeat v where
  rpeatEvery v :: (TimingInterval v) (MTask v t) -> MTask v t | type t
  rpeat      :: (MTask v t) -> MTask v t | type t
```

Listing 4.4: Function to create the `rpeatEvery` task.

4.2.5 Alternative notations

An alternative notation for tuning the refresh rate was considered. This notation included a new operator called the tune operator. The tune operator—defined in Listing 4.5—is based on a similar operator in `iTask`. This operator can be applied to all tasks, but it only affects tasks where it makes sense. In a task where the customization does not make sense, such as the `delay` task, is the custom refresh rate ignored. This could be confusing for the programmer, and it is difficult to detect where the refresh rate is ignored as the custom refresh rate does not directly influence the outcome of the program. This problem could be solved by restricting the tasks on which the tune operator can be applied. However, this requires the introduction of a new phantom type. All things considered, the tune operator works unintuitively or makes the implementation of the DSL more complex.

```
class (<<@.) infixl 2 v :: (MTask v a) (Option v) -> MTask v a | type a
```

Listing 4.5: Definition of the tune operator.

4.2.6 Implementation

Task tree nodes are decorated with a refresh rate. This refresh rate is by default equal to the default value and is patched at runtime if a custom refresh rate is assigned to this node. This required two new bytecode instructions, namely `TuneRateMS` and `TuneRateSec`. These instructions receive the upper and lower bound of refresh rate from the stack and patch the last created task node with this interval. Note that all `TimingInterval` constructors can be expressed using these instructions as they all express an interval with an upper and lower bound. For instance, `ExactSec (lit 1)` translates to into `[1, 1]`. The use of new bytecode instructions offers two advantages over extending the already existing instructions. The program’s size is only increased if the user specifies a custom refresh rate and the number of changes to the instruction set is minimized.

4.3 Deriving refresh rates

The evaluation interval of a task is equal to its derived refresh rate when there are no unevaluated parts of the task. Both the evaluation interval and refresh rate are derived at run time and on the device. The refresh rate is determined by walking through the task tree recursively. The refresh rate of the top node in the task tree is the refresh rate used as evaluation interval. The derived refresh rate of a task node is not always equal to the user-defined refresh rate since the refresh rate of, for example, the `rpeatEvery` task influences the restart time of the body and not the rate at which the body is evaluated.

The process of deriving a refresh rate of a task tree is formally defined by the function $\mathcal{R} : TaskTree \rightarrow [Int, Int]$. The definition of the function \mathcal{R} is shown in Figure 4.1.

$$\begin{aligned}
\mathcal{R} : TaskTree &\rightarrow [Int, Int] \\
\mathcal{R}(t_1.||t_2) &= \mathcal{R}(t_1) \cap_{safe} \mathcal{R}(t_2) & (1) \\
\mathcal{R}(t_1.\&\&t_2) &= \mathcal{R}(t_1) \cap_{safe} \mathcal{R}(t_2) & (2) \\
\mathcal{R}(t \gg * . [c_1 \dots c_2]) &= \mathcal{R}(t) & (3) \\
\mathcal{R}(repeat\ t) &= \begin{cases} \mathcal{R}(t) & \text{if } t \text{ is unstable} \\ [r_1 - startTime, r_2 - startTime] & \text{otherwise} \end{cases} & (4) \\
\mathcal{R}(waitUntil^3 e) &= [e - LastWakeUp, e - LastWakeUp] & (5) \\
\mathcal{R}(rateLimit\ t) &= \mathcal{R}(t) & (6) \\
\mathcal{R}(t) &= \begin{cases} [\infty, \infty] & \text{if } t \text{ is stable} \\ [r_1, r_2] & \text{otherwise} \end{cases} & (7)
\end{aligned}$$

Figure 4.1: Definition of function \mathcal{R} . r_1 and r_2 are the upper and lower bound on the default or user-defined refresh rate of the task node in question.

$$X \cap_{safe} Y = \begin{cases} X \cap Y & X \cap Y \neq \emptyset \\ Y & Y_2 < X_1 \\ X & otherwise \end{cases}$$

Figure 4.2: Definition of safe intersection on intervals

Parallel combinators The first two lines of the function \mathcal{R} define the refresh rate for the `.||.` and `.&&.` combinators. These parallel combinators combine two tasks into a new task to evaluate them in parallel. The refresh rate of the parallel task node is defined as the safe intersection of the child task nodes' refresh rates. The definition of a safe intersection is shown in Figure 4.2. The conventional intersection is not sufficient because it yields an empty intersection if both intervals are disjoint. The safe intersection always returns a non-empty interval. The resulting refresh rate is, by definition of the safe intersect, an interval within both refresh rates whenever possible. The refresh rate of the task node with the earliest interval is returned if the intersection is empty. Hence, one of the task nodes is evaluated more often, but none of the task nodes less than it should. Evaluating a task node more often is safe, but it should be avoided when possible.

Sequential combinators Line 3 defines the refresh rates of the sequential combinator. The refresh rate of this combinator is equal to the refresh rate of the left-hand side. The right-hand side is not considered since that side will not evaluate in the succeeding evaluation. The right-hand side replaces the combinator and left-hand side in the evaluation when the step condition is

³`waitUntil` is internally used by the `delay` node to wait until it is later than a specified time. The `delay` node does not require a separate definition in \mathcal{R} because it is changed into the `waitUntil` node during the first evaluation.

met. Before the step condition is met, only the left-hand side is evaluated. All other sequential combinators are derived from this combinator and do not require a separate entry in \mathcal{R} .

Rpeat and RpeatEvery Line 4 defines the refresh rate of the `rpeat` task node. The `rpeatEvery` task uses the same task node as the `rpeat` task node but with a patched refresh rate. The derived refresh rate of the `rpeat` task node is equal to the refresh rate of the inner task node if the inner task node is unstable. Otherwise, the derived refresh rate is equal to the default or the user-defined refresh rate minus the time when the body restarted. If the previous iteration was overdue, this start time is counted from the time the previous iteration became overdue. The resulting refresh rate can never become negative since the minimal value of the lower and upper bound of the refresh rate is capped to zero. The default refresh rate of the `rpeat` task is $[0, 0]$ (see Table 4.1). The child is under those circumstances directly restarted when it becomes stable.

Delay Line 5 defines the interval for the `waitUntil` task node that is internally used to create delays. The `waitUntil` task waits until the current time is greater than the time specified as an argument. This task returns the time until the delay is over as unstable value and the time it overshoot as a stable value after the delay has passed. Although the value constantly changes during the delay, it is considered safe to sleep until the delay is over since this is the behaviour a user would expect. The refresh rate of the `waitUntil` task node is thus equal to the wait time left.

Other basic tasks Line 7 defines the refresh rate of all other basic task nodes. These nodes have an infinite refresh rate if the task node is stable and a fixed default refresh rate (see Table 4.1) in all other instances. This default refresh rate can be overridden by a custom user-defined refresh rate if the task supports it. Task nodes with a stable value have an infinite refresh rate because their value never changes. Extra evaluations of a stable task node are thus unnecessary. Basic task nodes with a non-zero lower bound on the refresh interval are wrapped into a special and newly introduced task node. This `RateLimit` task node limits the evaluations of the underlying node to no more than the lower bound of the refresh rate allows. Without this rate-limiting node, the lower bound on the refresh rate is just a suggestion instead of a hard limit. It is important to notice that the `RateLimit` task node is for internal use only and there is no corresponding language construct to wrap tasks in it by hand. The following example that monitors a plant is used to explain the `RateLimit` task node in more detail:

```
monitorPlant :: Main (MTask v Bool) | mtask, dht, LightSensor v
monitorPlant
= declarePin MoistureSensorPin PMInput \m->
  lightsensor (i2c 0x23) \l->
    let moistureSensor =
        readA` (ExactSec (lit 300)) m
        >>*. [IfValue ((<=.) (lit 100)) (\_. rtn (lit True))]
    in
    let lightSensor =
        light l
        >>*. [IfValue ((<=.) (lit 150.0)) (\_. rtn (lit True))]
    in {main = lightSensor .||. moistureSensor }
```

The light sensor in this example should be polled between 0 and 100 ms and the moisture sensor every 5 minutes. This poses a problem because it is not possible to partially evaluate a task tree. The parallel operator chooses the refresh rate of the task node with the earliest deadline. That is, in this example, the refresh rate of the light sensor task. Consequently, the moisture level of the soil is measured too often. The user could have important reasons for limiting the polling rate. The sensor could have a limited lifetime, or the measurement could be slow. This is when the `rateLimit` node comes into play because it limits the evaluation rate for the child node. This node emits the stored value of the inner task node until it is allowed to reevaluate the child node again. The refresh rate of the `rateLimit` node is equal to the refresh rate of the child as defined on line 6.

Task	Default Interval
tvoc	[0, 2000]
co2	[0, 2000]
temperature	[0, 2000]
humidity	[0, 2000]
gesture	[0, 1000]
AButton	[0, 100]
BButton	[0, 100]
readA	[0, 100]
readD	[0, 100]
light	[0, 100]
motion	[0, 100]
soundPresence	[0, 100]
soundLevel	[0, 100]
getSds	[0, 2000]
rpeat	[0, 0]

Table 4.1: Default refresh rates

4.3.1 Implementation

The calculation of the refresh rate is, apart from some technical details, implemented as described in the function definition of \mathcal{R} . This calculation occurs after the task's evaluation is finished, but only if the task contains no unevaluated parts. When there are unevaluated parts, the evaluation interval is equal to **[0, 0]** and otherwise to the refresh rate of the task tree root. Whether a task contains unevaluated parts is determined during evaluation and stored as a property of it. The property also stores information about whether it is necessary to recalculate the evaluation interval and corresponding refresh rate. This is an optimization to avoid recalculations for unchanged tasks. The resulting evaluation interval expresses a time interval for the next evaluation of this task. This interval is used by the scheduler, explained in Chapter 5, which selects the most suitable time within this interval.

4.3.2 Examples

In Appendix B are various examples and their corresponding evaluation intervals explained in detail.

Chapter 5

Task scheduling on the device

The previous chapter explains how the evaluation interval of tasks is derived. Tasks should be evaluated within this time interval, but it is up to the scheduler to select the most optimal time to evaluate a task. This scheduler is part of the RTS and schedules the tasks at runtime. The scheduler is, besides choosing when and which tasks are evaluated, also responsible for using the created sleep time as power efficiently as possible. This sleep time could be spent by just keeping the device in active mode but switching to one of the other power states is in most cases more efficient.

5.1 Introducing the scheduling problem

5.1.1 Objectives

The device can run multiple tasks simultaneously. New tasks are received in the communication phase of the event loop and tasks are removed when they are stable or deleted by the server. Every task has bounds on the time of the next evaluation. The primary goal of the scheduler is to use the least energy while evaluating all tasks within their bounds. This goal is defined more concretely by the following objectives:

- **Meet the deadline** All tasks need to be started before the deadline whenever possible. No real-time guarantees are provided, but the scheduler provides its best effort to evaluate tasks in time.
- **Long sleep times are preferred over short sleep times** Less frequent longer sleep periods are preferred over more frequent short sleep periods. More efficient power states can be utilised if the device sleeps for a longer period of time. For example, the ESP8266 can only switch to light sleep if the sleep period is longer than the Delivery Traffic Indication Message (DTIM) interval. Due to this objective, the next evaluation time is expressed as an interval instead of a single point in time. An interval gives the scheduler more flexibility to perform multiple evaluations consecutively. Resulting in longer sleep times and a lower power consumption.
- **Evaluate tasks only when necessary** This objective entails postponing task evaluations as long as possible as evaluating a task before the deadline could lead to extra and unnecessary evaluations. This objective conflicts with the previous objective that prefers to bundle task evaluations to create longer sleep times. The resulting scheduler must therefore

have a fine balance between those two objectives. There are also non-conflicting instances of unnecessary evaluations that should be avoided. For instance, if there are two tasks. One with `[0, 2000]` as evaluation interval and the other task with `[0, 10000]` as evaluation interval. In this instance, an evaluation round must occur every 2 seconds to meet the deadline of the first task. It is allowed to evaluate the second task every 2 seconds too, but that leads to unnecessary evaluations and the sleep times are not prolonged. The preferred behaviour is that the first task is evaluated every 2 seconds and the second task every 10 seconds.

- **The power state with the least overall power consumption is activated during sleep** The previously explained objectives result in an as long as possible sleep time between task evaluations. This sleep time should be used optimally by selecting the power state that results in the lowest power consumption.
- **No starvation** All tasks ready to evaluate should be evaluated within a reasonable amount of time. Starvation can, for example, occur when a task uses the full capacity of the device and is given priority over other tasks. Other tasks are consequently never evaluated, which is undesirable.

Another non-functional requirement for the scheduler is reducing the overhead to a minimum. The scheduler runs on devices with limited processing power so it is crucial to make the scheduler as light-weight as possible.

5.1.2 Concepts

Three essential concepts in the scheduler need an introduction. *Absolute evaluation interval*, *evaluation round*, and *power state threshold* are explained in the following paragraphs.

Absolute evaluation interval The (relative) evaluation interval is an upper and lower bound on the next time a task should evaluate. This interval is defined relative to the evaluation time, which is meaningless if it is unclear when it was calculated. The absolute evaluation interval is, on the other hand, defined relative to the system time. The relative evaluation interval is easily transformed to an absolute evaluation interval by adding the time of the last evaluation to the relative evaluation interval. This makes the absolute evaluation interval independent of the evaluation time.

The absolute evaluation interval also prevents a time drift. Time drift is unwanted because it makes the evaluation time less precise. The effects of the time drift are illustrated in Figure 5.1. The figure shows a task with an evaluation interval of `[5, 5]`. Figure 5.1a shows the timeline without taking the evaluation time into account, causing a time drift. The second evaluation starts in this case after 8 ms instead of the desired 5 ms as the evaluation of the task took 3 ms. Figure 5.1b shows the same task, but the evaluation time is, in this case, calculated relative to the time of the evaluation. The task is now evaluated exactly every 5 ms. It is important to notice that the next evaluation time of a task is defined relative to its last evaluation. Hence, a time drift can still occur if a task is evaluated earlier or later than its deadline. This is, in most cases, not problematic, but it is often solvable by wrapping the task in a `repeatEvery` task if it causes problems for a specific use case. The `repeatEvery` task corrects for overdue tasks because it defines its next restart time relative to the time it should have been restarted instead of the actual restart time. Preventing earlier task evaluations can be done by using a refresh rate with an equal lower and upper bound.

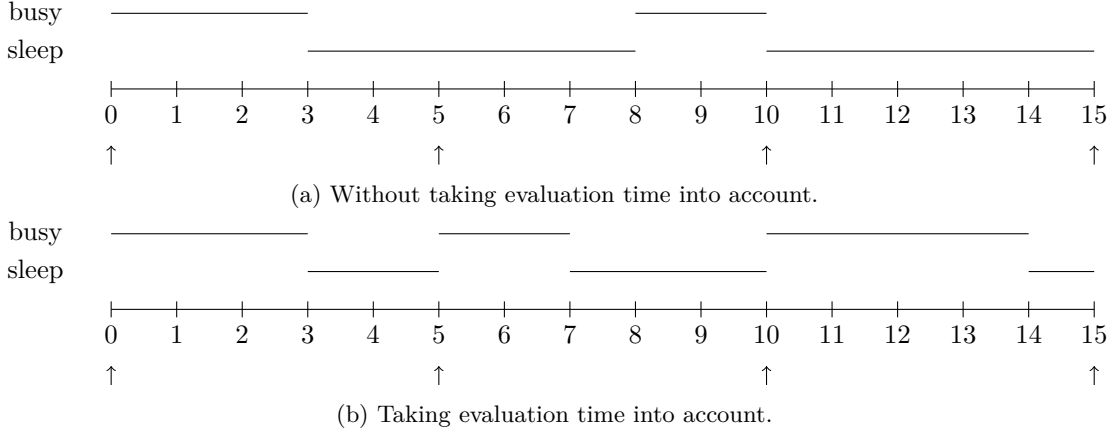


Figure 5.1: Visualisation of the time drift when the evaluation time is not taken into account.

Evaluation round The event loop repeatedly executes the following phases: communication, evaluation, garbage collection, and sleep. The scheduler determines in the evaluation phase which tasks to evaluate and in which order. One execution of this evaluation phase is called an evaluation round. A task is evaluated at most once every evaluation round to ensure short and, in most cases¹, finite evaluation rounds. Tasks are broken up into small steps, and only one of those steps is carried out in an evaluation. The opposite strategy is carrying out evaluation steps until there is no work left. This race-to-sleep strategy leads to unnecessary evaluations and less sleep time. For instance, a task that repeatedly reads a sensor with a refresh interval of `[0, 2000]` is ready to evaluate directly after the previous evaluation. It is in that case, desirable to sleep two seconds before the task is evaluated again. A race-to-sleep algorithm leads, in this case, to infinite evaluations in one evaluation round. Preventing multiple evaluations of a task in an evaluation round also ensures that the communication and garbage collection steps are evaluated regularly and in a timely manner.

Power state threshold The power state threshold (PT_x) is a precalculated value used to resolve the optimal power state for a given sleep time. The value indicates the break-even point when the power state becomes more efficient than the one-step lighter power state. A lighter power state often has less switching overhead, but it has, on the other hand, a higher base power consumption. The formula to calculate the power state threshold for power state x is:

$$PT_x = \underset{n \in \mathbb{N}}{\text{minimize } n} \text{ such that } (n - TT_{prev(x)}) * P_{prev(x)} + TP_{prev(x)} \geq (n - TT_x) * P_x + TP_x$$

subject to $n \geq MT_x$

This formula finds the minimum time n where the power consumption of the one-step lighter power state ($prev(x)$) is greater or equal to the power consumption of power state x . The power consumption of a power state is calculated as the base power consumption (P_x) multiplied by the time in that power state. This time is equal to the sleep time minus the time to transition to that state (TT_x). The resulting power consumption is increased with the overhead (TP_x) to switch to that power state and back to the active state. Finally, the resulting power state threshold is bounded by the minimum time (MT_x) to enable a power state. This calculation is inspired by the way EARTH selects its power state.

¹Infinite expressions can still lead to an infinitely long evaluation round.

The power state threshold for the ESP8266 and ESP32 is irrelevant because the activation of sleep modes is offloaded to the underlying firmware. Power state selection could also (partly) be done by the mTask RTS, but offloading the power state selection to the firmware is more power efficient². The information required to determine the power state thresholds for other devices is gathered experimentally.

For instance, the mTask implementation of the Adafruit Feather M0 WiFi has multiple power states: Active, Idle with WiFi connection, Standby with WiFi Connection, Idle without WiFi connection and Standby without WiFi connection. Active mode is used if the device is not sleeping. Standby with WiFi connection has a power state threshold of 1 second since that is the minimum time to enable that power state. Next, standby without WiFi connection has an experimentally determined power state threshold of approximately 14.7 seconds. Lastly, idle mode is used when the wait time is less than a second or when the device is waiting for interrupts. Idle mode is enabled when there are interrupts since standby mode stops the `millis` timer. Thus, it is unclear how long the device was asleep when it wakes up. This leads to some technical problems in the implementation. Therefore, it is decided to use a lighter sleep mode when there are interrupts. The power state threshold to use idle mode without WiFi is 15.6 seconds.

The values to determine the power state threshold are gathered experimentally by measuring the power consumption and transition time during the (transition of) the power state. This experiment is repeated ten times, and the average values are used to calculate the power state threshold. The resulting values are an approximation of the actual values since they vary depending on the environment. For instance, the time to reconnect to the WiFi network depends on the type of access point and distance to it.

5.1.3 Problem relaxations

The task scheduling problem is too difficult to solve efficiently. It requires, for example, knowledge about the evaluation time of tasks. This information is not available and challenging, or maybe even impossible to determine. The problem is, therefore, relaxed by simplifying two sub-problems.

The evaluation time of tasks is unknown It is impossible to know the exact evaluation time of tasks without executing them. Solving this sub-problem would require a solution to the halting problem, which is known to be unsolvable. Maybe it is possible to approximate the evaluation time of tasks in some cases due to the well-defined and, in most cases, known-to-be finite rewrite steps, but this is outside the scope of this thesis.

This problem is solved by freezing the time at the start of every loop iteration. The time during one individual task evaluation was already frozen because the absolute evaluation interval uses the start time of the task instead of the finishing time. This time freeze is extended to a complete evaluation round with possibly multiple task evaluations to simplify the problem. In that case it is no longer necessary to know the evaluation time of a task. Tasks are with this simplification evaluated on time if the evaluation round started on or before the deadline. The deadline could, in practice, still be missed due to other task evaluations in the same evaluation round. For example, if two tasks should be evaluated within two seconds. It is valid to wait two seconds and freeze the time afterwards. Now it is possible to evaluate both tasks in time because the time is frozen. The last task evaluation could, in practice, be too late because the evaluation of the first task took some time. The last-to-evaluate task could be overdue but under normal

²Selecting a power state from mTask requires the device to be disconnected from the WiFi. Moreover, enabling light sleep for a limited time is currently not working as expected.

circumstances, never overdue by a massive amount of time as task evaluations are kept as short as possible. Freezing the current time is done by storing the time at the start of the event loop iteration and passing it to all tasks as the current time.

It is unknown when interrupts occur Perfect information about the occurrences of interrupts is not available because interrupts can, per definition, occur at any time. However, the scheduler needs this information to select the most optimal power state. A power state is selected based on the available sleep time of the device since a deeper power state often has a one-time penalty compensated by a lower base power consumption. It could be more efficient to pay the penalty upfront for lower overall power consumption in the long run. An example of such a penalty is the power spike caused by reconnecting to the WiFi network after wake up. Deciding whether this penalty is worth it is dependent on the time the power state is activated. Unfortunately, this information is not known upfront since an interrupt can cut the sleep time short at any time.

This problem is a variant of the ski-rental problem [20]. This problem considers the trade-off between renting skies for a small continuous fee or buying skies for a larger upfront fee. This problem is easy to solve if the number of days you are going to ski is known upfront, but the problem becomes more difficult if it is unknown for how long you will be skiing. Several strategies are available for this problem, but it is solved in mTask by simplifying the problem and ignoring interrupts in determining the sleep time.

5.2 Scheduling algorithm

The scheduling algorithm is split into two parts. The first part runs in the evaluation phase of the mTask loop and determines which tasks to evaluate and the order of evaluation. The second part runs in a newly created phase at the end of the mTask loop and determines the sleep time and power state.

5.2.1 Scheduling task evaluations

The evaluation of tasks is scheduled using a non-preemptive Earliest Deadline First (EDF) algorithm implemented as a priority queue. The tasks in this queue are sorted from the earliest to the latest deadline, which is the same as the upper bound of the absolute evaluation interval. Tasks with the same deadline are placed in the queue in the order they were enqueued. The algorithm pops the next-to-evaluate task from the queue and reinserts it after evaluation if it did not become stable. This process continues until one of the following statements about the next to evaluate task T is true:

- **Task queue is empty** This is the case if all tasks are stable or if there are no tasks yet.
- **Task T was already evaluated during this evaluation round** This condition prevents multiple evaluations of a task in an evaluation round. Disallowing multiple evaluations of tasks is required to make the evaluation time of an evaluation round finite and short. Not limiting the maximum number of evaluations in an evaluation round leads to long or even infinite evaluation rounds when a task is ready to evaluate directly after evaluation again. Take, for example, a simple task that continuously reads a digital pin. This task could be evaluated multiple times or even indefinitely in an evaluation round because the lower bound on the default refresh rate is zero. Greedy evaluation of this task would thus lead to an infinite evaluation round which then again causes problems with the handling of

communication and garbage collection.

Determining whether a task has already been evaluated is done by checking whether the current time is equal to the last evaluation time. This works because the time is frozen at the start of the event loop.

- **The current time is smaller than the lower bound on the absolute evaluation interval of task T** This condition prevents the evaluation of tasks that are not allowed to evaluate yet. This condition also prevents unuseful earlier evaluations of tasks because those tasks have a non-zero lower bound to prevent extra evaluations. An example of a task node that is unuseful to evaluate earlier is the `rateLimit` task node. This task node always returns the same result until the underlying task is allowed to evaluate again. It is thus unuseful to check the result of the task before the wait time is over.

The resulting algorithm to perform one evaluation round is presented as an activity diagram in Figure 5.2.

5.2.2 Scheduling power states

The next iteration of the `mTask` event loop is scheduled in the last and newly added loop phase. Postponing the next iteration to a later moment saves energy by sleeping until it is time to evaluate the next iteration. The next iteration should start when the task with the earliest deadline is due. In other words, the device will wait until the earliest upper-bound of all absolute evaluation intervals. This deadline is obtained by inspecting the task in the front of the queue. The task at the front of the queue is, by definition of EDF ordering, the task with the earliest deadline. In the case of an empty queue, the sleep time is equal to a user-adjustable constant. The idea of procrastinating tasks as long as possible is based on LC-EDF. The resulting procedure is illustrated in Algorithm 1.

Algorithm 1 Determine sleep time.

```

1: function DETERMINESLEEPTIME
2:   if task queue is empty then
3:     return DEFAULT_SLEEP_TIME                                ▷ This is an adjustable constant
4:   else
5:     next ← peekNextTask()
6:     return deadline(next) - now

```

The next step is to select the most efficient power state that is available based on the previously determined sleep time. A power state can be unavailable when a particular device state prevents the enabling of a power state. Finding the most efficient power state is done by selecting the deepest available power state with a power state threshold lower than the sleep time. More formally, the power state for a given sleep time (n) is defined as follows:

$$\text{power state} = \underset{p \in \text{States}}{\text{find deepest power state such that } PT_p \leq n \text{ and } p \text{ is available}}$$

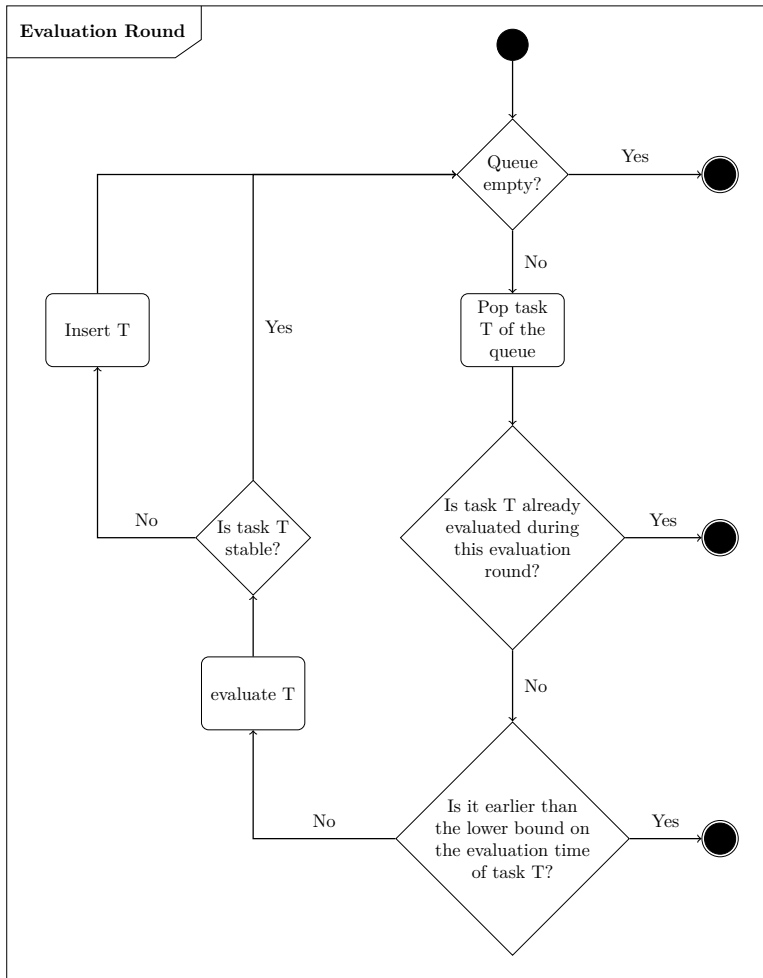


Figure 5.2: Activity diagram of the evaluation part of the scheduling algorithm.

5.3 Properties of the scheduling algorithm

The resulting algorithm meets the four functional and one non-functional requirement set for the algorithm. The following paragraphs describe why and how the scheduler satisfies these requirements.

Meeting the deadline All tasks meet their deadlines under the relaxation that the time is frozen during an evaluation round. Real time can not be frozen; hence, tasks can miss their deadline in practice by a pinch. The scheduler guarantees, despite this limitation, that the device never sleeps past a task deadline.

Preferring long sleep times over short sleep times Tasks are bundled together as much as possible to create longer sleep periods. This is done by evaluating all available tasks until one of the stop conditions is met.

Preventing unnecessary evaluations Finding a suitable balance between this objective and the previous objective is challenging. While designing this scheduler, it is decided to evaluate as many tasks as possible until one of the stop conditions is met. The next iteration of the event loop is thereafter postponed as long as possible since the sleep time is equal to the time until the next deadline. This balance seems suitable as tasks evaluations are less costly than short sleep times. For instance, reading a sensor value costs less energy than reconnecting to the WiFi network.

Other instances of unnecessary evaluations that are not conflicting with the long sleep times objective are prevented since the scheduler is based on EDF scheduling. These instances arise when a task is evaluated, and there are tasks with an earlier deadline not allowed to evaluate. In other words, the task evaluation could be postponed to the next evaluation round. EDF scheduling evaluates tasks, by definition, from early to later deadline. Thus, if the current task is evaluated too early, all subsequent tasks are also evaluated too early. Hence, a task is not evaluated unnecessarily if the scheduler evaluates tasks in order of the deadline. Take, for example, the two tasks T1 and T2 with a refresh rate of $[0, 1000]$ and $[0, 2000]$ respectively. The queue during the evaluation of these tasks is illustrated in Figure 5.3. The prevention of unnecessary evaluations by EDF scheduling is clearly visible in the evaluation round at one second. It is, at this time, only necessary to evaluate task T1 because T2 can be evaluated at the next evaluation round at two seconds. This behaviour is achieved using EDF scheduling because task T1 is placed back on the front of the queue, and T2 is therefore not evaluated during this evaluation round. It is important to notice that the behaviour in practice could be different due to slight imperfections in the sleeping mechanism. The wake-up at one second could, for example, be 1 ms late, which makes the next absolute evaluation time of task T1 equal to $[1001, 2001]$. This changes the behaviour because the deadline of task T2 is now earlier than the deadline of task T1. Task T2 is, in that case, evaluated in the evaluation round at one second.

Optimal power state selection This subproblem is simplified by assuming that perfect information about the duration of sleep is available. However, this information is in practice unknown since interrupts can occur at any time while sleeping. Anyhow, the most optimal power state is selected if the upfront determined sleep duration is not interrupted.

Preventing starvation No starvation occurs because the EDF ordering is based on the absolute evaluation time which incorporates the time of the last evaluation. To put it another way, tasks evaluated longer ago are automatically given priority over more recent tasks. This aging mechanism ensures that all tasks get evaluated within a reasonable time. It is also important to notice that task evaluations are divided into small steps. This eliminates the need for preemption to interrupt slow or long tasks. Figure 5.4 illustrates the evaluation of two tasks with a refresh interval of $[0, 0]$. This shows that both tasks are evaluated, and no starvation occurs due to the time differences between evaluation rounds.

Complexity The resulting algorithm can be implemented efficiently using a priority queue. Selecting the task to evaluate can be done in $\mathcal{O}(1)$ and (re)inserting a task in $\mathcal{O}(n)$ whereby n is the number of tasks in the queue. It is even possible to implement reinsertion in constant time using a Fibonacci heap [21], but a naive approach is preferred due to the small number of tasks in practice.

Determining the sleep time is also done in $\mathcal{O}(1)$, since only the first item in the queue needs to be inspected. Finding the resulting power state is slightly more complex because multiple power states must be considered. However, this is still efficient due to the small number of power states.

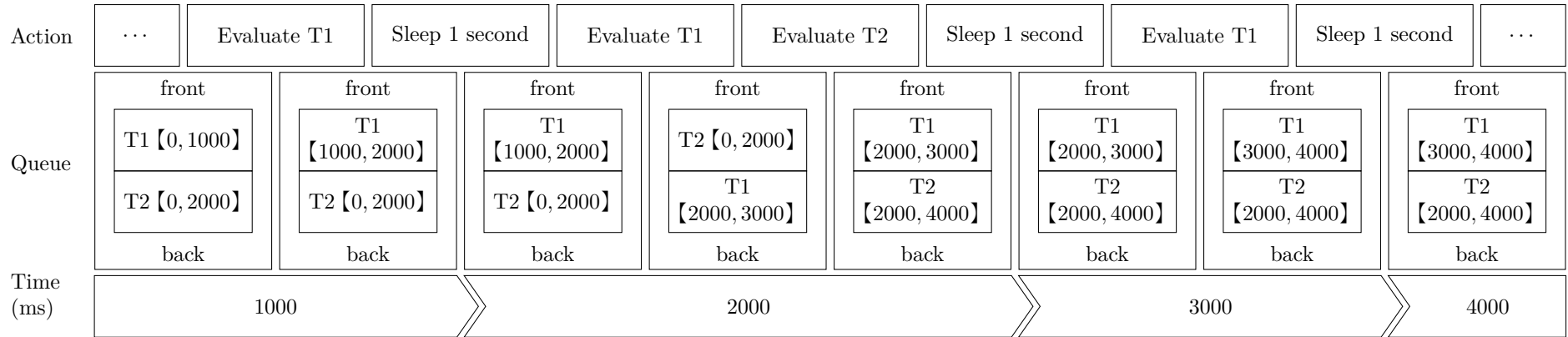


Figure 5.3: Illustration of the contents of the queue during the evaluation of two tasks. Task T1 has a refresh rate of $[0, 1000]$ and Task T2 has a refresh rate of $[0, 2000]$. The interval show represents the absolute evaluation interval of the tasks.

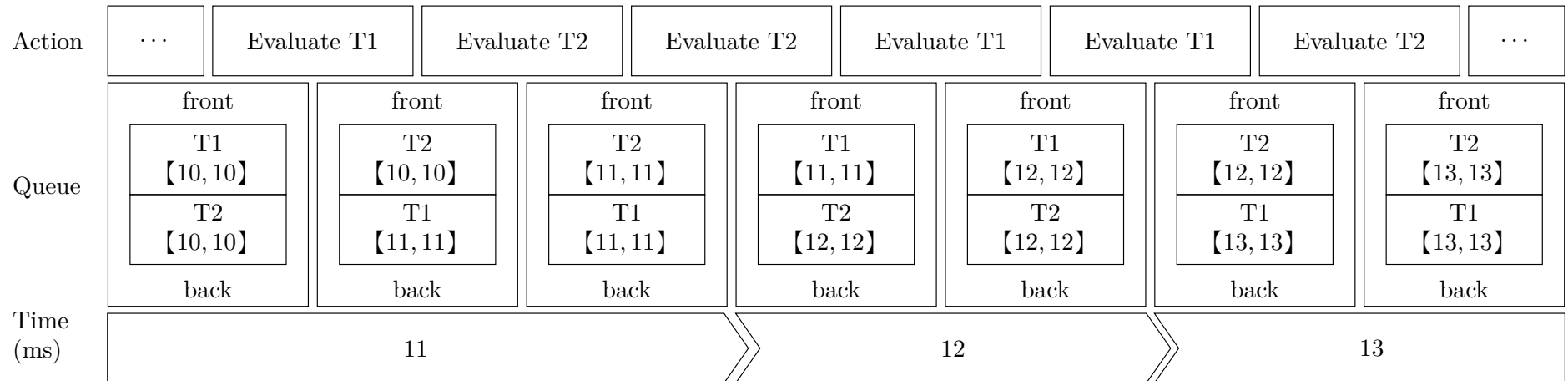


Figure 5.4: Illustration of the contents of the queue during the evaluation of two tasks. Task T1 has a refresh rate of $[0, 0]$ and Task T2 has a refresh rate of $[0, 0]$. The time it takes to evaluate one evaluation round is assumed to be 1 ms. The interval show represents the absolute evaluation interval of the tasks.

Chapter 6

Interrupts

Interrupts are interruptions of the normal control flow to handle pressing events. When an interrupt triggers, a special kind of function—called the Interrupt Service Routine (ISR)—is executed. The ISR handles the event and the normal control flow is continued afterwards. Special about interrupts is that the pin is monitored in hardware. Hence, the ISR is executed directly after the trigger and no processing power is required to monitor the pin.

Tasks in mTask are not evaluated in the ISR. Instead, the ISR is merely used to mark the task for execution. The actual evaluation of the task is postponed to a later moment. Directly evaluating the task in the ISR causes problems because the program state could not be ready for it. Moreover, it is a best practice to keep the ISR short. This approach also has drawbacks, namely that mTask cannot provide guarantees about the time between the event and the evaluation of the interrupted task anymore. Some typical applications of interrupts, such as reading serial data from a digital pin, do not work as expected. That is not a major drawback because mTask is not designed for such low-level applications. Besides, interrupts still offer two advantages over regular tasks:

- **Better energy usage** Repeatedly polling an input pin requires the device to wake up periodically to read the value. Interrupts improve the energy consumption by allowing the device to sleep until the value of that pin changes. There are also some sensors, such as the CCS811 air quality sensor¹ with support for interrupts. This sensor has the option to fire an interrupt when a value exceeds a critical limit.
- **Fewer missed events** The device may be sleeping or evaluating other tasks when an event such as a button press occurs. In such a case, the event can be over again before the device has checked the presence of that event. Interrupts solve this problem by interrupting the current activity to set a flag in the RTS to indicate that the interrupt has occurred. It is now clear for the task associated with the interrupt that the event took place. It is important to notice that multiple events occurring shortly after each other could still lead to missed events because only the last one is stored. I.e. the RTS only knows if an interrupt has occurred and not how many times it occurred.

¹See <https://www.sciosense.com/wp-content/uploads/documents/SC-001232-DS-2-CCS811B-Datasheet-Revision-2.pdf>

6.1 Interrupt types

A variety of events such as timers, completed SPI transfers, and completed ADC conversions can trigger interrupts. Most of the times, these types of interrupts are too low level for mTask. Therefore, only one relevant type of interrupt is implemented, namely interrupts triggered by events on the GPIO pins. Table 6.1 gives a complete overview of the interrupt types supported in mTask.

Type	Triggers when
Change	the state of the input changes
Falling	the state of the input becomes low
Rising	the state of the input becomes high
Low	the state of the input is low
High	the state of the input is high

Table 6.1: Overview of the interrupts types.

6.2 Interrupts on the Arduino platform

An example is used throughout this chapter to illustrate interrupts. This section introduces this example by providing an implementation in Arduino C(++) for the Adafruit Feather M0.

```
#define LEDPIN 13
#define INTERRUPTPIN 11
#define DEBOUNCE 30
volatile byte state = LOW;
volatile unsigned long last = 0;
volatile bool cooldown = true;

void setup() {
  pinMode(LEDPIN, OUTPUT);
  pinMode(INTERRUPTPIN, INPUT);
  LowPower.attachInterruptWakeup(INTERRUPTPIN, buttonPressed, RISING);
}

void loop() {
  LowPower.sleep();
  digitalWrite(LEDPIN, state);
  delay(DEBOUNCE);
  cooldown = false;
}

void buttonPressed() {
  if (!cooldown)
    state = !state;
  cooldown = true;
}
```

This light example switch toggles the onboard LED connected to pin 13 as soon as the user presses the button. In other words, the light switches on after a button press if it was previously off and vice versa. The device sleeps between button presses. This functionality is implemented using the Low Power library².

6.3 Modelling interrupts

This section explains how interrupts are modelled in mTask. Different ways to model interrupts were considered, but defining a task to emit the status of the interrupt was, in the end, the best solution. The definition of the `interrupt` task is show in Listing 6.1.

```

:: InterruptMode
= Change
| Rising
| Falling
| Low
| High

class interrupts v where
  interrupt :: InterruptMode (v p) -> MTask v Bool | pin p

```

Listing 6.1: Definition of the `interrupt` task.

The function to create the `interrupt` task has two arguments. The first argument is of the type `InterruptMode` and indicates the type of interrupt. The second argument is the pin to which the interrupt is attached.

The resulting task emits no value until the interrupt is triggered and the status of the associated pin as a stable value afterwards. An interrupt task only listens for interrupts until it is triggered. It does not reset itself after the task value was observed. The `repeat` task can be used to reset the task if it is required to listen to the same interrupt multiple times.

One of the advantages of interrupts is the improved energy consumption. Polling, as with the `readD` task, requires the device to wake up periodically. Interrupts listen for events while the device is sleeping and do not require periodic wake-ups. The refresh rate of an interrupt task is thus equal to $[\infty, \infty]$. When the interrupt triggers, the refresh rate is changed to $[0, 0]$. This happens in the ISR and ensures that the triggered task is evaluated in the next evaluation round. The triggered `interrupt` task is transformed into a `stable` task with a refresh rate of $[\infty, \infty]$ after evaluation. Figure 6.1 defines the refresh rate of `interrupt` tasks formally.

$$\mathcal{R}(\text{interrupt}) = \begin{cases} [0, 0] & \text{if it is triggered} \\ [\infty, \infty] & \text{otherwise} \end{cases}$$

Figure 6.1: Extension of the function \mathcal{R} for interrupts.

²See <https://www.arduino.cc/en/Reference/ArduinoLowPower>

Alternative ways to model interrupts Other considered ways to model interrupts are step continuations, tasks as interrupt handlers, and implicit interrupts.

Modelling interrupts as a new type of step continuation feels intuitive at first glance since `iTask` models exceptions in the same way. The idea was to treat an interrupt as a variant of an exception that propagates through the task. However, one crucial difference is that an exception originates from the task on the left-hand side of the combinator, but an interrupt does not. This often resulted in an empty task on the left-hand side of the step operator. For example, a simple task that waits for an interrupt looks like this `rtrn () >>*. [OnInterrupt (Rising d2 High) handle]`. An alternative solution to prevent this problem was to let interrupts originate from a `readD` task. However, it is in the current way `mTask` evaluates tasks not possible to propagate the interrupt elegantly.

Another considered solution is to model interrupts using interrupt routines. An interrupt routine is a task that is evaluated when the associated interrupt fires. This task is added in parallel to the original task after firing. This option was disregarded because the interrupt routine is an isolated task which makes communication between the original task and interrupt routine difficult.

The last considered solution was making interrupts implicit. Interrupts are, in this approach, automatically used in places where it is possible. For instance, a `readD` task that is waiting for a high value can be transformed to an interrupt that triggers when the value becomes high. A disadvantage of this approach is the lack of control over when and where an interrupt is used.

6.3.1 Examples

This section introduces two examples to demonstrate the use of the `interrupt` task in practice. The first example shows an `mTask` implementation of the earlier introduced light switch. The second example illustrates the advantages of using interrupts in combination with sensors.

Light switch This example implements the light switch introduced in Section 6.2. The starting point of this task is the `switch` function. This function starts with setting the value of `pin` with the built-in led to `x`. Variable `x` represents the status of the LED. Thereafter, a small delay is inserted to debounce the button. The `interrupt` task follows after the debounce and has no value until the button is pressed. After the button press, the function `switch` is executed again but with the negation of `x`. By comparing the Arduino C(++) implementation in Section 6.2 with the `mTask` implementation, it becomes clear that (1) the `mTask` implementation requires no global state and (2) the programmer no longer has to deal with ISRs and their quirks. As a result, the `mTask` implementation is more elegant and compact than the Arduino implementation.

```
lightSwitch :: Main (MTask v Bool) | mtask v
lightSwitch
  = declarePin ButtonPin PMInput \button->
    declarePin BuiltinLEDPin PMOutput \led->
      fun \switch=(\x->
        writeD led x
        >>|. delay (lit 50) // Debounce
        >>|. interrupt Falling button
        >>|. switch (Not x)
      )
  In {main=switch (lit False)}
```


Air quality sensor Some sensors, such as the CCS811 Air Quality sensor, have a special pin for triggering interrupts. The sensor fires this interrupt when a configurable threshold is exceeded. Utilising interrupts for this sensor is, after implementing `setTvocLimit`, implementable using just a macro definition.

```
setTvocLimit :: (v AirQualitySensor) (v Int) -> MTask v ()
tvocLimit sensor limit pin
  := setTvocLimit sensor limit >>|. interrupt Falling pin
```

6.4 Handling interrupts

Interrupt tasks, like other tasks, have their own type of nodes in the task tree. Interrupt tasks differ from other tasks in multiple places. First, interrupts must be enabled or disabled in hardware during the creation and deletion of tasks. Secondly, interrupt tasks are triggered by interrupts that can occur at any time. The RTS should therefore be able to handle the occurrence of an interrupt during every phase of the event loop.

6.4.1 Registering and deregistering interrupts

An interrupt is enabled in hardware when the first task waiting for that kind of interrupt—e.g. combination of pin and interrupt type—is created. Conversely, the interrupt is disabled again when the last interrupt task of that kind is deleted. Interrupts should be disabled when there are no tasks waiting for that kind of interrupt because unused interrupts can lead to unwanted wake ups, and only one kind of interrupt can be attached to a pin. Enabling and disabling interrupts is done in a general way in which tasks register themselves after creation and deregister after deletion. Comparable with a create and destroy hook. The registration of interrupt events is updated in these hooks too. The event registration and hooks are kept general to allow the system to utilise this hook for subscribing to other events such as SDS updates.

Event registration A record is kept of tasks that are waiting for events. This registration has two advantages. First, tasks waiting for an event can easily be located. Secondly, it can be used to quickly deduce whether the interrupt should be enabled or disabled in hardware. Since an interrupt only needs to be enabled if it is the first task waiting for it, there is no need to re-enable the interrupt if another task registers the same interrupt. On the other side, an interrupt can only be disabled when there are no more tasks waiting for it. An alternative to the event registration is to scan all tasks and their task trees for interrupt task nodes when an interrupt triggers or to deduce whether an interrupt can be disabled. The use of an event registration is, in the end, preferred over scanning all tasks.

The registration consists of an event identifier and task identifier. Each event has a unique identifier to describe the trigger. For example, the falling interrupt on pin 5 has a unique identifier to find all tasks waiting for that specific interrupt quickly. The task identifier refers to the task waiting for the interrupt. An identifier is used instead of a pointer because the garbage collector could change pointers.

This event registration is stored as linked lists of task tree nodes. Events are stored as a task tree node so that the garbage collector cleans up unused events.

Enabling and disabling interrupts Enabling and disabling interrupts on the Arduino-supported devices is done using the Arduino Application Programming Interface (API). Enabling or disabling interrupts without the Arduino API works differently for each device. Fortunately, the Arduino API abstracts this away, so it works the same way for each device. What does vary from device to device are the supported interrupt types and pins. The support of the interrupt is checked before enabling the interrupt and an `mTask` exception is thrown if it is unsupported. An exception is also thrown if another type of interrupt is already attached to the pin, as it is not possible to attach two interrupt types to the same pin.

6.4.2 Triggering interrupts

The task triggered by the interrupt is not immediately evaluated in the ISR, because it is not always safe to evaluate the task. It could, for example, be the case that the garbage collector is interrupted, which leads to incorrect pointers. Additionally, the ISR should be kept as short as possible. This problem is solved by storing the information of the interrupt and postponing the processing of the interrupt to the normal control flow.

The processing of interrupts happens at the start of the event loop. This is directly after wakeup if the device was asleep because the `mTask` loop is always executed after wake up. The processing does not happen immediately when an interrupt occurs in the middle of an iteration. Interrupt processing in `mTask` is not time critical, and finishing the iteration takes, in general, a short amount of time. Cutting an iteration short is possible but not needed and therefore not implemented due to time constraints. Instead, the processing of interrupts is postponed to the next iteration. The device does not sleep in between those iterations because there is a guard in place to prevent sleeping while there are unprocessed interrupts.

Interrupts are processed by looking in the event registry for tasks subscribed to the interrupt. The task tree of each subscribed task is searched for task nodes waiting for the triggered interrupt. A flag is set and the status of the pin is stored in those nodes. Storing the status of the pin is needed because the action (e.g. a pulse) that triggered the interrupt can be over before the task is evaluated. Reading the status of the pin during the evaluation can thus be too late. Furthermore, it prevents resource-intensive readings as the status of the pin is in some interrupt types already known.

Next, the evaluation interval of the task is set to `[0, 0]`, and the task is inserted at the front of the scheduling queue. This ensures that the task is evaluated in the next evaluation step of the event loop.

Finally, the event is removed from the registration and the interrupt is disabled. The interrupt can be disabled as all tasks waiting for the interrupt become stable after firing. More occurrences of the interrupts do not change the value of the task as stable tasks keep the same value forever. Therefore, it is no longer necessary to keep the interrupt enabled, and it is relatively cheap to enable it again if needed in the future.

6.4.3 Evaluating interrupt tasks

Evaluating the interrupt task node in the task tree is straightforward because most of the work was already done when the interrupt was triggered. The task emits the status of the pin as a stable value if the information in the task shows that it was triggered. Otherwise, no value is emitted.

Chapter 7

Resulting power reductions

In this chapter, the resulting power savings achieved by the adaptations to mTask are presented. First, the methodology used to gather these results is explained. Next, the power consumption of several representative programs is illustrated and analyzed using electrical current graphs.

7.1 Methodology

The power measurements used throughout this chapter are measured using an INA219 current sensor¹. This sensor is configured to measure currents from 0 to 400 mA with a resolution of 0.1 mA. The exact details of this configuration can be found in the Adafruit INA219 library². An Arduino Uno controls this sensor and measures the current every two milliseconds. These measurements are then processed and transformed into graphs by a personal computer. All currents are measured on the power lines of the USB cable powering the Adafruit Feather M0 WiFi³. An illustration of the used setup is shown in Figure 7.1. The total power consumption over a period is calculated by integrating using the composite trapezoidal rule. This power consumption calculation is, among other things, used to calculate the lifetime of battery-powered devices with a 1200 mAh battery. Finally, the sensors used in the example programs are the Wemos SHT30 shield⁴ to measure the temperature and the Wemos BH1750 shield to measure the intensity of the ambient light.

¹See <https://www.adafruit.com/product/904>

²See https://github.com/adafruit/Adafruit_INA219

³The old situation is measured on the mTask version of 23 March 2021 (Client commit: 090c5aec and Server commit: 8e2e8bc2). This version did not support the Adafruit Feather. Therefore, this version is adapted to support this development board. The automatic power-saving mode of the WiFi module is enabled in this version to make the power savings achieved by the novel execution model clearer. The use and implementation of this power mode is trivial and requires only one line of code. No power modes of the microcontroller were utilised in the old implementation.

⁴`Wire.begin()` must be commented out in the Wemos SHT3x library to use this sensor in combination with the Adafruit Feather. This is caused by a bug in the Wemos library.

Task	Average energy consumption		Difference
	Old	New	
Blink	111.3 mW	52.1 mW	-53%
Thermometer	204.2 mW	28.8 mW	-86%
Light switch ¹	147.3 mW	108 mW	-27%
Thermometer & Plant Monitor	190.2 mW	22.5 mW	-88%

¹ Notice that this result is dependent on the ratio between the time spent waiting for interrupts and time spent processing interrupts. The device consumed on average 46.8 mW while waiting for interrupts, and it consumed on average 365.68 mW to process the button press.

Table 7.1: Average energy consumption of the example programs during the displayed measurements.

7.2.1 Blink

Blinking an LED is one of the more often used examples in the world of microcontrollers. The code to create a task that blinks the onboard LED with a frequency of 1 second is shown below. This code was already introduced in Section 3.1.1 and works on both the old and new version of mTask.

```
blink :: Main (MTask v Bool) | mtask v
blink
  = declarePin BuiltinLEDPin PMOutput \led->
    fun \blink=(\x->
      delay (lit 1000)
      >>|. writeD led x
      >>=. blink o Not)
    In {main=blink (lit True)}
```

Figure 7.2a and Figure 7.2b show the current draw of the blink task in both the old and new situation. The x-axis shows the time in milliseconds and the y-axis the current. A higher current draw means that the device is consuming more energy. The background colour of the graph is dependent on the device state. In the red areas, the device is in active mode and in the green areas the device is sleeping. In the old situation, the device stays in active mode throughout the complete execution. The old execution model evaluated tasks at a continues rate and did not utilise the sleep modes of the microcontroller. As shown in Figure 7.2b, the new execution model only evaluates the task when the LED state is changed. The device is in sleep mode in between these toggles, recognizable by the lower current draw. There are still some current spikes while sleeping, as indicated by the two arrows. These spikes are caused by the WiFi modem waking up to receive the DTIM. In the end, the adaptations to mTask reduce the energy consumption by 53% during the measured time. This translates to an improvement in battery life from 2.2 to 4.9 days.

7.2.2 Thermometer

The temperature in a room changes gradually and relatively slow. An application to measure the room temperature and show it in an online dashboard can therefore limit the refresh rate of the temperature sensor to reduce the power consumption. The task below implements the device part of such an application. This task measures the temperature every minute and sends the value to the server. The refresh rate is specified as an argument in the `temperature` task. An equivalent task without a refresh rate replaces this task during the measurements of the old situation.

```
measureTemp :: Main (MTask v Real) | mtask, dht v
measureTemp =
  DHT (DHT_SHT (I2C (UInt8 0x45))) \dht->
    {main = temperature` (BeforeSec (lit 60)) dht}
```

The graphs in Figure 7.3a and Figure 7.3b plot the current consumption of this digital temperature meter. The `measureTemp` task is only evaluated every minute in the new situation due to the adapted refresh rate. Contrary to the old situation where the task evaluates at maximum speed. The current consumption of the device reflects this as the device cannot utilise the sleep modes. Moreover, many current spikes are visible caused by the device transmitting new values over the WiFi network. In the new situation, the device sleeps and disables the WiFi connection between readings. This results in a current consumption below the measurable threshold of 0.1 mA between temperature measurements. The device wakes up every minute, represented by the spikes in the graph, to reconnect to the WiFi network and measure the temperature. The power consumption is in the new situation reduced by 86% compared with the old situation. This translates into an improvement of battery life from 1.2 to 8.7 days based on the measured period. Even more substantial power savings and battery life improvements can be accomplished by increasing the interval to measure the temperature.

7.2.3 Light switch

This example implements a smart light switch. It toggles the LED when the user pushes the button. Button presses are infrequent since users typically leave the light on or off for a long period. These button presses are short lived and can happen at any time. Hence, interrupts are, for this application, the best way to monitor the button state. A device with interrupts enabled can continue monitoring the button state while sleeping. Another approach is to check the button state at a high enough frequency to minimize the risk of missing button presses. The latter approach requires the device to stay active. The two tasks below implement this light switch in these two ways. The first implementation is the same as introduced in Section 6.3.1 and monitors the button using interrupts. The second implementation monitors the button by continuously checking its state. The second implementation is used in the measurements of the old situation since interrupts were added to the `mTask` language in this thesis.

```

// Light switch based on interrupts
lightSwitch :: Main (MTask v Bool) | mtask v
lightSwitch
  = declarePin ButtonPin PMInput \button->
    declarePin BuiltinLEDPin PMOutput \led->
      fun \switch=(\x->
        writeD led x
        >>|. delay (lit 50) // Debounce
        >>|. interrupt Falling button
        >>|. switch (Not x)
      )
    In {main=switch (lit False)}

// Light switch based on continously checking the button state
lightSwitch :: Main (MTask v Bool) | mtask v
lightSwitch
  = declarePin ButtonPin PMInput \button->
    declarePin BuiltinLEDPin PMOutput \led->
      fun \switch=(\x->
        writeD led x
        >>|. delay (lit 50) // Debounce
        >>|. readD button
        >>*. [IfValue Not (\_.switch (Not x))]
      )
    In {main=switch (lit False)}

```

Figure 7.4a and Figure 7.4b show the current draw of the light switch during its execution. Halfway through the measurements is the button pressed to switch the LED on. This toggle is in Figure 7.4b, clearly visible by the spike in current caused by the device reconnecting to the WiFi network. The toggle is hard to recognise in the old situation but happens around 8 seconds since the start of the measurements. This also shows the differences between the approach with and without interrupts. In the new situation, the device stays asleep until it has work to do. Conversely, the device monitors the state of the button actively in the old situation.

The graph also shows a drawback of this particular interrupt implementation. Button presses are slightly less responsive in the implementation with interrupts as the device reconnects to the WiFi network before the LED state is changed. This could be improved by evaluating the task before the reconnect, but that comes with several challenges. One of these challenges is determining how many evaluation rounds are needed before the WiFi network can be reconnected without decreasing responsiveness. Due to time constraints is the implementation of this left for future work.

7.2.4 Plant monitor & Thermometer

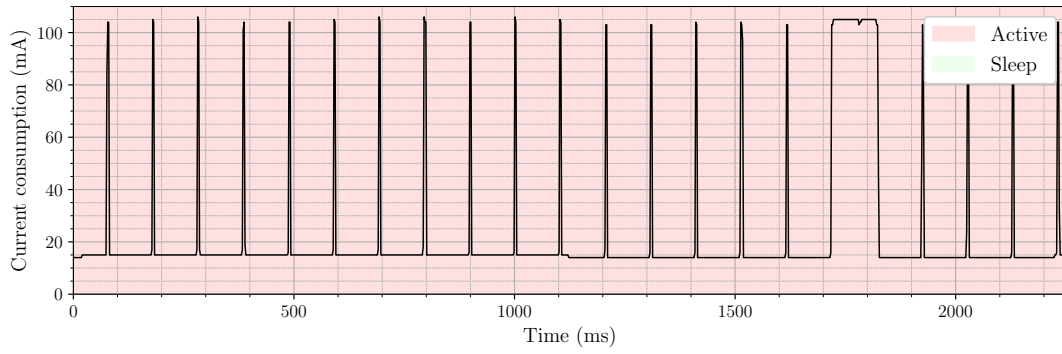
An mTask device can run multiple independent tasks simultaneously. This example utilises this functionality to combine a plant monitor and thermometer into one device. The plant monitor task is already introduced in Section 4.3, but the sensors have, in this case, a different refresh rate. This task observes a plant's environment and becomes stable when one of the monitored values exceeds a threshold. It monitors two values, the amount of water in the soil and the light intensity. The value of the soil sensor should be read at least every 5 minutes and the light

intensity at least every 90 seconds. The implementation of this task is shown below. The `readA`` and `light`` task are replaced with the task variant without refresh rate while conducting the measurements of the old situation. The thermometer task that measures the temperature every minute is the same as explained earlier.

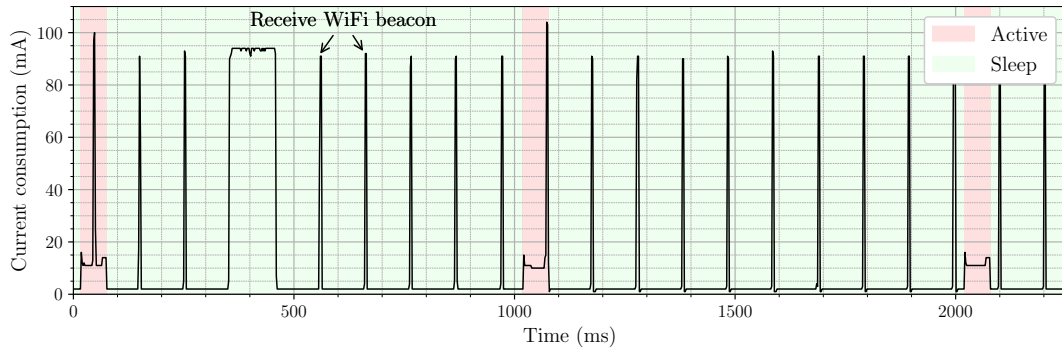
```
monitorPlant :: Main (MTask v Bool) | mtask, dht, LightSensor v
monitorPlant
  = declarePin MoistureSensorPin PMInput \m->
    lightsensor (i2c 0x23) \l->
      let moistureSensor =
          readA` (BeforeSec (lit 300)) m
          >>*. [IfValue ((<=.) (lit 100)) (\_. rtn (lit True))]
        in
          let lightSensor =
              light` (BeforeSec (lit 90)) l
              >>*. [IfValue ((<=.) (lit 150.0)) (\_. rtn (lit True))]
            in {main = lightSensor .||. moistureSensor}
```

Figure 7.5a and Figure 7.5b show the current consumption of the device while running these two tasks. These tasks are constantly evaluating in the old situation. In the new situation, the device evaluates the tasks every minute and sleeps the rest of the time. Both tasks are evaluated every minute since the scheduler combines both evaluations. The `monitorPlant` task should evaluate at least every 90 seconds because that is the refresh rate of the `.||.` combinator combining both plant sensor measurements. This combinator uses, per definition of \mathcal{R} , the intersection of the child tasks' refresh rates. Resultingly, the `measureTemp` task should evaluate at least every minute and the `monitorPlant` task should evaluate at least every 90 seconds. The scheduler combines these two evaluations to prevent extra wake-ups.

Consequently, the graph of the current consumption is roughly comparable with the current consumption graph of the thermometer task. This example also shows that the three previous examples are representative for more complex tasks. Tasks with a fast refresh rate result in the behaviour as in the blink example. Similarly, tasks with a more prolonged refresh rate result in the behaviour as in the thermometer example. Lastly, tasks with interrupts show comparable behaviour with the light switch example.

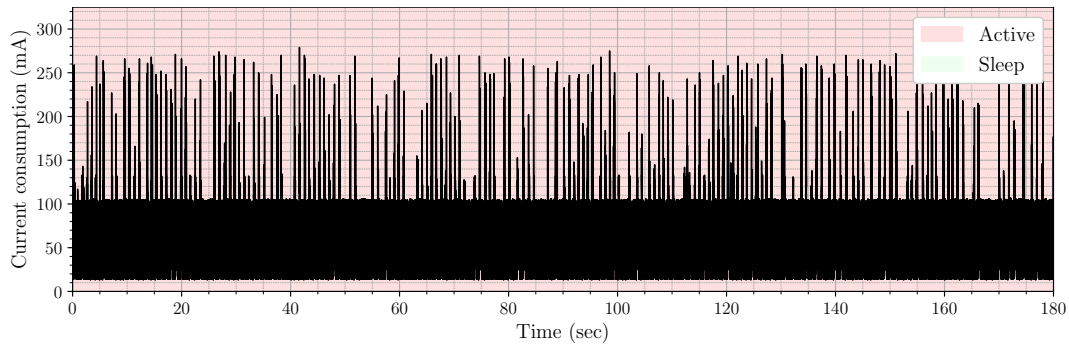


(a) Old situation

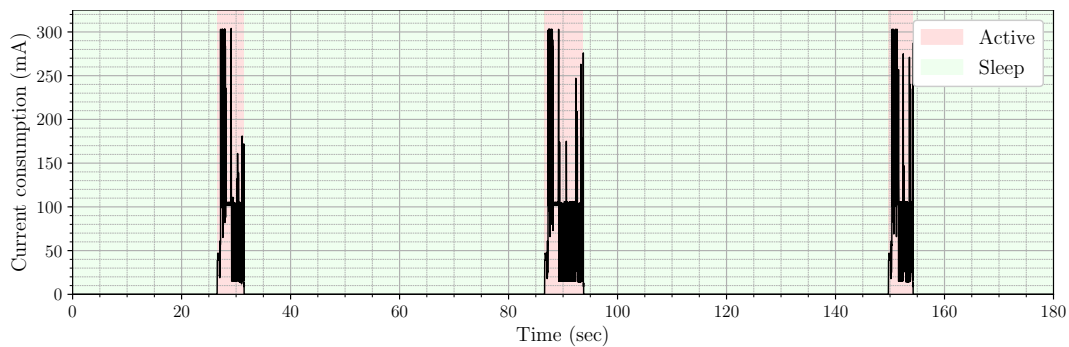


(b) New situation

Figure 7.2: Current draw of the device during the execution of the blink task. Figure 7.2a shows the situation before any modifications were made and Figure 7.2b shows the situation afterwards.

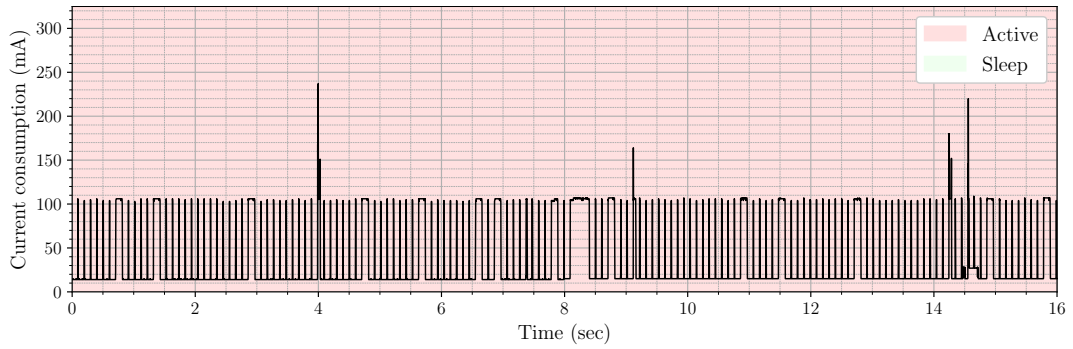


(a) Old situation

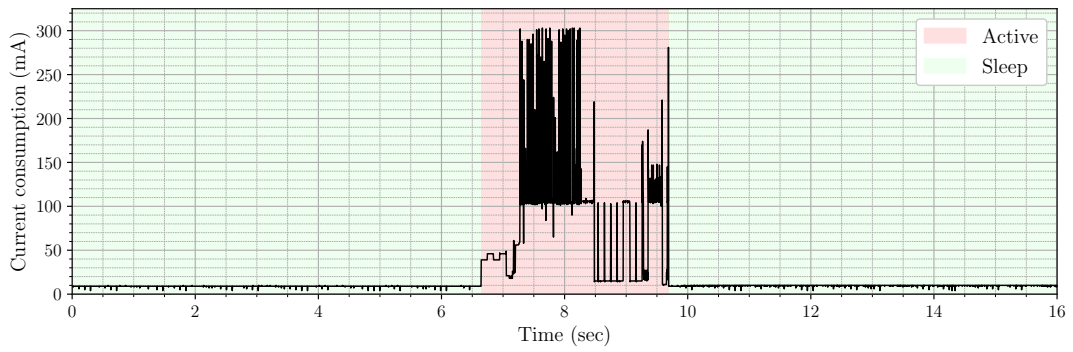


(b) New situation

Figure 7.3: Current draw of a thermometer task during its execution. Figure 7.3a shows the situation before any modifications were made and Figure 7.3b shows the situation afterwards.

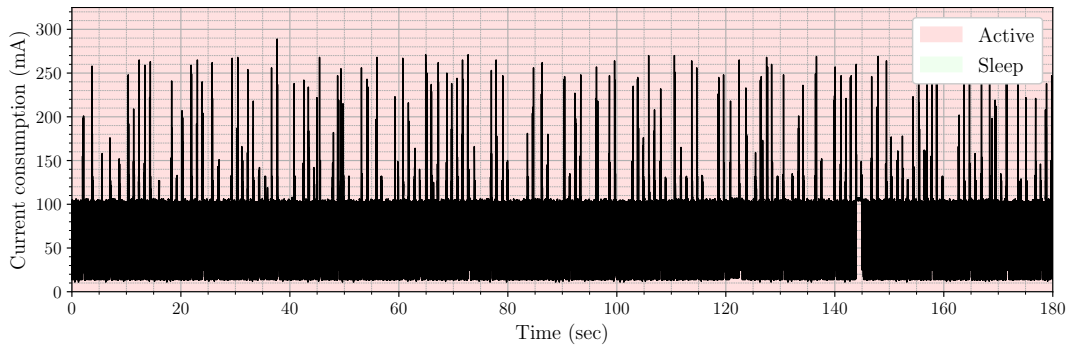


(a) Old situation

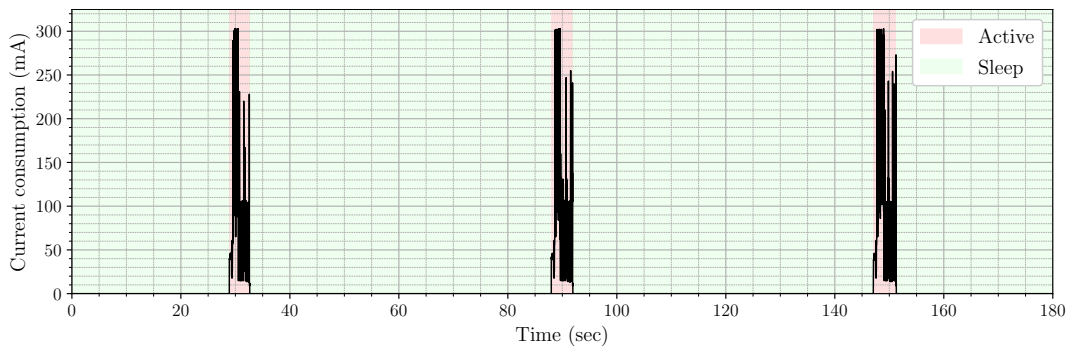


(b) New situation

Figure 7.4: Current draw of the light switch task. Figure 7.4a shows the situation before any modifications were made and Figure 7.4b shows the situation afterwards.



(a) Old situation



(b) New situation

Figure 7.5: Current draw of the device when it is running the `measureTemp` and `monitorPlant` task simultaneously. Figure 7.5a shows the situation before any modifications were made and Figure 7.5b shows the situation afterwards.

Chapter 8

Conclusion

8.1 Conclusion

The goal of this thesis was to reduce the energy consumption of devices running IoT applications developed using TOP. Reducing the energy consumption is better for the environment and improves the longevity of battery-powered devices. This study shows that the energy consumption of mTask programs can be decreased with minimal changes to the language and at an abstraction level that suits the TOP paradigm. The essence of the presented energy-saving approach is to minimize the number of task evaluations and use the freed-up time to sleep. The number of task evaluations is reduced in two ways: by introducing a novel execution model and using interrupts.

Dynamic evaluation frequency The former version of mTask evaluated all tasks as often as possible. A fast evaluation pace is not always necessary. For instance, it is sufficient to evaluate a task that measures the temperature of a room every minute since a room temperature changes gradually. Therefore, the novel mTask execution model evaluates tasks based on events instead of continuously. At the start, this execution model only uses clock based-events to trigger task evaluations. This change required a way to calculate the next evaluation time of a task based on its state. The scheduler uses the derived bounds to select the most optimal time to evaluate a task.

These changes to the mTask system remain hidden from the programmer and are backwards compatible. Nevertheless, the language is extended to let the user specify the refresh rate of tasks where the evaluation frequency is dependent on the use case (e.g. tasks reading a sensor value). The fact that only minimal changes were required to the mTask language shows that the abstractions of mTask—and TOP in general—are suitable for handling low-level sleeping functionalities automatically and without requiring extra effort from the programmer.

Interrupts Checking a sensor value at a reduced rate is more efficient than constantly checking its value, but it can be made even more energy efficient using interrupts. Interrupts allow the device to sleep while monitoring sensors. An extra benefit is that interrupts are better in detecting short-lived events such as button presses compared to polling. The mTask language and system are therefore extended with GPIO interrupts.

An analysis of the electrical current during the execution of several example programs shows that the method presented in this thesis reduces energy consumption substantially. Moreover, this study demonstrates that it is possible to automatically utilise power states of microcontrollers in

programs written using the TOP paradigm. The abstraction level of the demonstrated approach suits TOP and requires minimal to no extra effort from the programmer.

8.2 Related work

This section discusses several works related to this thesis. The first subsection discusses how iTask, the other TOP language, prevents unwanted evaluations. The subsections after that discuss the Functional Reactive Programming (FRP) paradigm and embedded operating systems. Lastly, other notable systems to program microcontrollers are examined.

8.2.1 iTask

The iTask language—by design—prevents unnecessary evaluations as it evaluates tasks solely based on events. Additionally, SDSs have a powerful notification system to trigger evaluations when an SDS changes. The foundation of this event system is a `select` call on the TCP socket. This allows the system to hand over control to the operating system while still being able to react to events. Somewhat comparable with interrupts in mTask that allow the device to sleep while monitoring events. The way iTask prevents unnecessary evaluations differs from mTask since iTask is shallowly embedded in the host language. Hence, it is in iTask not possible to do an intensional analysis on the task tree. Event-based evaluations are also easier to implement in a shallowly embedded language as they have access to the features of the host language. Another reason for the difference between the mTask and iTask languages is the purpose for which they are designed. In the mTask language, many tasks are based on polling and benefit from time-based events. Thus, it is more suitable for mTask to derive the next evaluation time from a task tree instead of using a memory-intensive event registration. Nonetheless, some of the primitives introduced in this thesis, such as the rate-limiting concept, could be beneficial to iTask. For instance, to limit the rate of sensor readings in the iTask sensor library¹.

8.2.2 Functional Reactive Programming

FRP [22] is a programming paradigm that is roughly comparable with TOP. Nevertheless, there are also many differences between the paradigms, as shown in a comparison by Stutterheim et al. [23]. The FRP paradigm programs hybrid systems at a high level using events and behaviours. Events represent discrete event occurrences and behaviours values that vary over time. Multiple FRP implementations for microcontrollers exists, such as Hae. Hae [24] is a deeply embedded DSL hosted in Haskell. This language preserves resources by updating values that vary over time at discrete time points instead of continuously. These time points can be defined by the user, similar to our approach. An important difference, besides the paradigm, is that an Hae program compiles into C++ code compared to mTask, which is more dynamic and interprets the program on the device.

Another notable research of Belwal et al. presents an energy-aware scheduler for Priority-based FRP [25]. The scheduling approach of this paper differs from ours as it uses a different technique to reduce the energy consumption. This technique is called Dynamic Voltage and Frequency Scaling (DVFS) which slows the speed of the microcontroller down instead of sleeping. DVFS is explained in more detail in Section 8.3.

¹See <https://gitlab.science.ru.nl/clean-and-itasks/sensors>

8.2.3 Embedded operating systems

There are many operating systems for microcontrollers developed over the years, like FreeRTOS [13], Mantis OS [26] and Nano RK [27]. Most of these operating systems have support for multithreading and energy management. Generally speaking, they differ from mTask in two fundamental ways. First, these systems are based on fixed programs in flash memory compared to mTask where tasks are created dynamically. Secondly, these operating systems operate at a lower abstraction level than mTask.

8.2.4 Other

Several other related systems are not yet discussed but worthwhile to examine. First, Firmata [28] is a generic protocol to control microcontrollers from a host computer. This protocol does not encode complete programs, but it consists of specific commands created by a host program. The firmware running on the devices then executes these commands. The Firmata protocol supports a sampling interval for analog and I²C data that is continuously reported back to the host, similar to the refresh rate of mTask programs. However, this sampling interval works at a global level and is not modifiable for individual sensors. Moreover, the standard Firmata implementation does not utilise the sleeping modes of microcontrollers.

Controlling an Arduino using this Firmata protocol from Haskell is implemented by the `hArduino`² DSL.

This DSL served as a starting point for Haskino [29], which no longer uses Firmata. Haskino and mTask share some similarities as they are both interpreted DSLs with a scheduler to run multiple jobs concurrently [30]. However, they are also very different as Haskino is imperative and low-level compared to mTask, which is high-level and designed using the TOP paradigm. Due to this low-level nature is an extra effort from the programmer required to insert pauses in the execution. The mTask RTS inserts a delay in various cases automatically and without requiring extra effort from the programmer. Besides, mTask is energy-aware and manages its power modes autonomously. The difference between a high and low-level language is also reflected in the interrupt implementations. In mTask, an interrupt only marks a task for evaluation, while Haskino executes the job inside the ISR.

Another notable project is ESPHome³. This project is not scientifically published but interesting to discuss as it shares similarities with the work presented in this thesis. ESPHome is a system to control an ESP-based microcontroller through a powerful configuration file. Although it is no programming language, it can create complex systems using conditionals, actions and triggers. Similar to mTask, the system can monitor multiple sensors with a different refresh rate simultaneously. It also uses a scheduler to schedule the execution of components and can use sleep modes. The main difference between ESPHome and mTask is that ESPHome generates applications using a configuration file and mTask specifies programs using a DSL. The mTask language can express more complex programs than the configuration file natively⁴ can. In addition, ESPHome converts the configuration file to an Arduino program compared to mTask, which interprets the program on the device.

²See <http://leventerkok.github.io/hArduino/>

³See <https://esphome.io/>

⁴The configuration file can also contain raw C++

8.3 Future work

The novel execution model proposed and implemented in this thesis significantly reduces the energy consumption of mTask, but there are still several points of improvements left for future work.

8.3.1 Hardware

The mTask system uses sleep modes to reduce the energy consumed by microcontrollers. However, sleeping is not always the most optimal way to use idle time. For some microcontrollers, a technique called DVFS or a combination of DVFS and sleep [31] is more optimal. DVFS lowers the power consumption by dynamically decreasing the clock frequency and voltage used to operate the processor when the full capacity of the CPU is not required to complete tasks within time. A lower frequency and optionally a lower voltage slows the execution speed down and lowers the device's energy consumption. Supporting DVFS is not trivial and requires multiple changes to mTask, but it is an interesting topic for future research.

Moreover, this study focuses on the sleep modes of microcontrollers, but some peripherals also have different power modes. The use of these power modes is left for future research due to time constraints.

8.3.2 Scheduler

The execution model uses a scheduler to schedule task evaluations in the most energy-efficient way. However, the scheduler does not provide real-time guarantees since the evaluation time of tasks is not taken into account. For future research, it could be interesting to examine if it is possible to provide real-time guarantees.

Bibliography

- [1] M. Rothmuller and S. Barker, “Iot ~ the internet of transformation 2020,” Juniper Research, Tech. Rep., 2020. [Online]. Available: <https://www.juniperresearch.com/whitepapers/iot-the-internet-of-transformation-2020>.
- [2] F. Xia, L. T. Yang, L. Wang, and A. Vinel, “Internet of things,” *International Journal of Communication Systems*, vol. 25, no. 9, pp. 1101–1102, Sep. 2012. DOI: 10.1002/dac.2417.
- [3] I. Charania and X. Li, “Smart farming: Agriculture’s shift from a labor intensive to technology native industry,” *Internet of Things*, vol. 9, p. 100 142, 2020. DOI: 10.1016/j.iot.2019.100142.
- [4] P. Sethi and S. R. Sarangi, “Internet of things: Architectures, protocols, and applications,” *Journal of Electrical and Computer Engineering*, vol. 2017, Jan. 2017. DOI: 10.1155/2017/9324035.
- [5] M. Lubbers, P. Koopman, and R. Plasmeijer, “Writing internet of things applications with task oriented programming,” in press.
- [6] R. Plasmeijer, P. Achten, and P. Koopman, “iTasks: executable specifications of interactive work flow systems for the web,” *ACM SIGPLAN Notices*, vol. 42, no. 9, pp. 141–152, 2007. DOI: 10.1145/1291220.1291174.
- [7] R. Plasmeijer, B. Lijnse, S. Michels, P. Achten, and P. Koopman, “Task-oriented programming in a pure functional language,” in *Proceedings of the 14th symposium on Principles and Practice of Declarative Programming*, 2012, pp. 195–206. DOI: 10.1145/2370776.2370801.
- [8] S. Michels and R. Plasmeijer, “Uniform data sources in a functional language,” *Submitted for presentation at Symposium on Trends in Functional Programming, TFP*, vol. 12, 2012. [Online]. Available: https://wiki.clean.cs.ru.nl/File:Sharing_Data_Sources.pdf.
- [9] P. Achten, P. Koopman, and R. Plasmeijer, “An introduction to task oriented programming,” in *Central European Functional Programming School*, Springer, 2015, pp. 187–245. DOI: 10.1007/978-3-319-15940-9_5.
- [10] T. Brus, M. van Eekelen, M. van Leer, and M. Plasmeijer, “Clean - a language for functional graph rewriting,” in *Conference on Functional Programming Languages and Computer Architecture*, Springer, 1987, pp. 364–384. DOI: 10.1007/3-540-18317-5_20.
- [11] M. Bambagini, M. Marinoni, H. Aydin, and G. Buttazzo, “Energy-aware scheduling for real-time systems: A survey,” *ACM Transactions on Embedded Computing Systems*, vol. 15, no. 1, Jan. 2016. DOI: 10.1145/2808231.
- [12] S. Zhuravlev, J. C. Saez, S. Blagodurov, A. Fedorova, and M. Prieto, “Survey of energy-cognizant scheduling techniques,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 24, no. 7, pp. 1447–1464, 2013. DOI: 10.1109/TPDS.2012.20.

- [13] R. Barry, *Using the FreeRTOS Real Time Kernel - A Practical Guide*. 2009.
- [14] Y.-H. Lee, K. Reddy, and C. Krishna, “Scheduling techniques for reducing leakage power in hard real-time systems,” in *15th Euromicro Conference on Real-Time Systems*, Jul. 2003, pp. 105–112. DOI: 10.1109/EMRTS.2003.1212733.
- [15] M. A. Awan and S. M. Petters, “Enhanced race-to-halt: A leakage-aware energy management approach for dynamic priority systems,” in *2011 23rd Euromicro Conference on Real-Time Systems*, 2011, pp. 92–101. DOI: 10.1109/ECRTS.2011.17.
- [16] M. Lubbers, P. Koopman, A. Ramsingh, J. Singer, and P. Trinder, “Tiered versus tierless iot stacks: Comparing smart campus software architectures,” in *Proceedings of the 10th International Conference on the Internet of Things*, 2020, p. 9. DOI: 10.1145/3410992.3411002.
- [17] J. Carette, O. Kiselyov, and C.-C. Shan, “Finally tagless, partially evaluated: Tagless staged interpreters for simpler typed languages,” *Journal of Functional Programming*, vol. 19, no. 5, pp. 509–543, 2009. DOI: 10.1017/S0956796809007205.
- [18] M. Lubbers, P. Koopman, and R. Plasmeijer, “Interpreting task oriented programs on tiny computers,” *Proceedings of the 31th Symposium on the Implementation and Application of Functional Programming Languages*, pp. 65–76, in press.
- [19] M. Lubbers, P. Koopman, and R. Plasmeijer, “Multitasking on microcontrollers using task oriented programming,” in *2019 42nd International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*, 2019, pp. 1587–1592. DOI: 10.23919/MIPRO.2019.8756711.
- [20] A. R. Karlin, M. S. Manasse, L. Rudolph, and D. D. Sleator, “Competitive snoopy caching,” in *27th Annual Symposium on Foundations of Computer Science (sfcs 1986)*, 1986, pp. 244–254. DOI: 10.1109/SFCS.1986.14.
- [21] M. L. Fredman and R. E. Tarjan, “Fibonacci heaps and their uses in improved network optimization algorithms,” *Journal of the ACM*, vol. 34, no. 3, pp. 596–615, Jul. 1987. DOI: 10.1145/28869.28874.
- [22] C. Elliott and P. Hudak, “Functional reactive animation,” in *Proceedings of the second ACM SIGPLAN international conference on Functional programming*, 1997, pp. 263–273. DOI: 10.1145/258948.258973.
- [23] J. Stutterheim, P. Achten, and R. Plasmeijer, “Maintaining separation of concerns through task oriented software development,” in *Trends in Functional Programming*, vol. 10788, 2018, pp. 19–38. DOI: 10.1007/978-3-319-89719-6_2.
- [24] S. Wang and T. Watanabe, “Functional reactive edsl with asynchronous execution for resource-constrained embedded systems,” in *Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing*. 2020, pp. 171–190. DOI: 10.1007/978-3-030-26428-4_12.
- [25] C. Belwal, A. M. K. Cheng, J. Ras, and Y. Wen, “Variable voltage scheduling with the priority-based functional reactive programming language,” in *Proceedings of the 2013 Research in Adaptive and Convergent Systems*, 2013, pp. 440–445. DOI: 10.1145/2513228.2513271.
- [26] S. Bhatti, J. Carlson, H. Dai, J. Deng, J. Rose, A. Sheth, B. Shucker, C. Gruenwald, A. Torgerson, and R. Han, “Mantis os: An embedded multithreaded operating system for wireless micro sensor platforms,” *Mobile Networks and Applications*, vol. 10, no. 4, pp. 563–579, 2005. DOI: 10.1007/s11036-005-1567-8.

- [27] A. Eswaran, A. Rowe, and R. Rajkumar, “Nano-rk: An energy-aware resource-centric rtos for sensor networks,” in *26th IEEE International Real-Time Systems Symposium*, 2005, pp. 256–265. DOI: 10.1109/RTSS.2005.30.
- [28] H.-C. Steiner, “Firmata : Towards making microcontrollers act like extensions of the computer,” in *Proceedings of the International Conference on New Interfaces for Musical Expression*, 2009, pp. 125–130. DOI: 10.5281/zenodo.1177689.
- [29] M. Grebe and A. Gill, “Haskino: A remote monad for programming the arduino,” in *Practical Aspects of Declarative Languages*, 2016, pp. 153–168. DOI: 10.1007/978-3-319-28228-2_10.
- [30] M. Grebe and A. Gill, “Threading the arduino with haskell,” in *Trends in Functional Programming*, 2019, pp. 135–154. DOI: 10.1007/978-3-030-14805-8_8.
- [31] G. Quan, L. Niu, X. Hu, and B. Mochocki, “Fixed priority scheduling for reducing overall energy on variable voltage processors,” in *25th IEEE International Real-Time Systems Symposium*, 2004, pp. 309–318. DOI: 10.1109/REAL.2004.23.

Acronyms

- ADT** Algebraic Data Type.
- AHB** Advanced High-performance Bus.
- APB** Advanced Peripheral Bus.
- API** Application Programming Interface.
- DPM** Dynamic Power Management.
- DSL** Domain Specific Language.
- DTIM** Delivery Traffic Indication Message.
- DVFS** Dynamic Voltage and Frequency Scaling.
- EDF** Earliest Deadline First.
- EDSL** Embedded Domain Specific Language.
- ERTH** Enhanced Race-To-Halt.
- FRP** Functional Reactive Programming.
- GPIO** General Purpose Input/Output.
- GUI** Graphical User Interface.
- IoT** Internet of Things.
- ISR** Interrupt Service Routine.
- LC-EDF** Leakage-Control EDF.
- MQTT** Message Queuing Telemetry Transport.
- RTC** Real-time clock.
- RTS** Runtime system.
- SDS** Shared Data Source.
- TCP** Transmission Control Protocol.
- TOP** Task-Oriented Programming.

Glossary

absolute evaluation interval Upper and Lower bound on the next evaluation time of a task relative to the system time.

evaluation interval Upper and Lower bound on the next evaluation time of the task relative to the previous evaluation. The evaluation interval is the same as the relative evaluation interval.

evaluation round One execution of the evaluation step in the mTask event loop.

power state threshold The power state threshold (PT_x) is a precalculated value to determine the optimal power state for a given sleep time. The value indicates the break-even point where the power state becomes more efficient than the one step lighter power state.

refresh rate Upper and Lower bound on the next evaluation time of the task tree relative to the previous evaluation of the task tree. It differs from the evaluation interval because a refresh rate is defined for task trees and the evaluation interval for tasks. Moreover, the evaluation interval is zero when there are unevaluated parts of the task.

relative evaluation interval Upper and Lower bound on the next evaluation time of a task relative to the previous evaluation.

Appendix

A Hardware Characteristics

IoT devices come in many forms, ranging from personal computers to low-cost ARM devices. These platforms all have different characteristics, such as power modes and ways to interact with the world. This appendix discusses all characteristics relevant to this thesis.

A.1 General

All hardware platforms have different characteristics. The more powerful devices such as personal computers with an operating system have more resources than low-level microcontrollers. On the other hand, programs running on microcontrollers have more control over features like sleep modes. This thesis mainly relies on two of these characteristics, namely power modes and interrupts.

Power modes

Most microcontrollers have several different power or sleep modes. These modes all have their own characteristics with regard to capabilities and power consumption. Example capabilities that differ between sleeping modes are the availabilities of timers and WiFi connectivity.

Interrupts

Interrupts are interruptions of the program execution by external events. The support of interrupts differs per device and depends on the underlying architecture. Several kinds of events can trigger interrupts, but the only relevant type for mTask are GPIO interrupts. Most of the microcontrollers with GPIO pins have support for interrupts on at least some of the pins.

A.2 Wemos Lolin D1 mini

The Wemos Lolin D1 mini is a development board based on the ESP8266⁵ microcontroller. Espressif manufactures this low-cost microcontroller with 112 KiB of memory and a clock speed of 80 or 160 MHz. The development board is equipped with WiFi, 11 digital IO pins and one analog pin.

⁵See https://www.espressif.com/sites/default/files/documentation/0a-esp8266ex_datasheet_en.pdf

Power modes

The ESP8266 has three different sleep modes⁶:

- **Modem sleep** This is the lightest of the three sleep modes. It keeps the CPU active but disables the WiFi module completely or in between Delivery Traffic Indication Message (DTIM) intervals. The access point periodically broadcasts a DTIM to inform the clients about a (possible) transmission of broadcast frames. The modem wakes up just before the DTIM is transmitted to receive the broadcasted frames. However, broadcasted data could be irrelevant to the device. It is, therefore, possible to configure the device so that it wakes up every n th DTIM, with a maximum interval of 10. Not waking up every DTIM transmission is more power efficient (see Table A.1), but the device can miss broadcasted data. Missing broadcast data has no consequences for the mTask implementation. Automatic modem sleep is enabled by default when the device connects to the router in station mode. Modem sleep mode can be enabled automatically by the firmware of the device when the device is idling. The firmware manages, in that case, the retrieval of the DTIM. It is also possible to enable this sleep mode by hand, but the WiFi connection is, in that case, closed.
- **Light sleep** This sleep mode works similar to modem sleep but disables—besides the modem—also the system clock. Additionally, the CPU is suspended. This means that the CPU stops its current tasks until the sleep time is over or wakes up by an interrupt. Light sleep can be enabled automatically by the device or enabled by hand using a system call. Automatically enabling light sleep happens when the device is waiting during, for example, a delay. The device keeps the WiFi connection in automatic light sleep open and wakes up to receive the DTIM. Similar to modem sleep.
- **Deep sleep** This is the deepest sleep mode available, and it uses the least power compared to the two other modes. The chip suspends all its functionalities in deep sleep mode, except for the Real-time clock (RTC). Hence, the program state is lost. The device can only wake up from this mode by pulling the reset pin to low. This can either be done by an external device or by the ESP8266 itself. The latter is done by physically connecting the reset pin and pin 16. GPIO pin 16 is pulled down by the RTC if the specified sleep time is over. It is not possible to keep the WiFi connected during deep sleep.

Item	Active	Modem sleep		Light sleep		Deep sleep
		Automatic	Forced	Automatic	Forced	Forced
WiFi connectivity	Connected	Connected	Disconnected	Connected	Disconnected	Disconnected
GPIO state	Defined	Undefined		Undefined		Undefined
WiFi	Transmitting / Receiving	Off		Off		Off
System clock	On	On		Off		Off
RTC	On	On		On		On
CPU	On	On		Pending		Off
Power usage (substrate current)	56-170 mA	15 mA		0.4 mA		~20 μ A
DTIM interval = 1	-	16.2 mA		1.8 mA		-
DTIM interval = 3	-	15.4 mA		0.9 mA		-
DTIM interval = 10	-	15.2 mA		0.55 mA		-

Table A.1: Overview of the sleep modes available on the ESP8266. The power usage for each DTIM interval is based on the average current (Adapted from Espressif⁷).

⁶See

https://www.espressif.com/sites/default/files/documentation/9b-esp8266-low_power_solutions__en.pdf

Automatic light sleep is the only supported sleep mode in mTask. Automatic light sleep results in the most optimal power savings while still preserving the program state and WiFi connection. A more substantial power reduction could be achieved by enabling deep sleep and storing the program state in non-volatile memory but implementing this is outside this thesis's scope. Moreover, non-volatile memory wears out fast if the program state is written to it at a high rate due to a limited amount of writing cycles. Another considered sleep mode was forced light sleep. Forced light sleep without an active WiFi connection results in a lower consumption for long sleep times. However, forced light sleep was ultimately not used due to known technical issues⁸.

Implementing automatic light sleep is straightforward since the device automatically sleeps when calling the `delay` function. It also requires that the device is in station mode with automatic light sleep enabled. These settings are programmatically set to the correct values and do not require extra steps of the user.

Interrupts

This development board supports five different types of GPIO interrupts on all digital pins except for pin D0. The five types of interrupts are: high, low, falling, rising and changing. The precise meaning of these interrupts is given in Table 6.1.

A.3 Arduino Uno

The Arduino Uno is the most used development board of the Arduino family. The board is based on the ATmega328P⁹ microcontroller from Atmel. It has a serial interface, 14 digital IO pins and 6 analog input pins. The ATmega328P is at the time of writing the microcontroller with the least resources that is supported by mTask. The microcontroller only has 2KB of RAM and a 16 MHz clock speed.

Power modes

The ATmega328P has six different sleep modes with a consumption ranging from 60 μ A to 1.9 mA. This is a notable reduction compared with the current consumption of 9.2 mA in active mode. However, the use of these sleep modes is currently not implemented in mTask.

Interrupts

Interrupts are in mTask only supported on pin D2 and D3. These are the only two pins with support for external interrupts. Other pins can also trigger interrupts, but these interrupts work on a group of pins instead of individual pins. Due to the extra bookkeeping required to find out which pin triggered the interrupt is decided to limit interrupts to pin D2 and D3.

⁷Adapted from Table 1-1 found on page 1 of https://www.espressif.com/sites/default/files/documentation/9b-esp8266-low_power_solutions_en.pdf. Additional data in table from https://www.espressif.com/sites/default/files/documentation/0a-esp8266ex_datasheet_en.pdf

⁸See <https://github.com/esp8266/Arduino/issues/6642>

⁹See https://ww1.microchip.com/downloads/en/DeviceDoc/Atmel-7810-Automotive-Microcontrollers-ATmega328P_Datasheet.pdf

A.4 Adafruit Feather M0 WiFi

The Adafruit Feather M0 WiFi is a development board with an ATSAM21G18¹⁰ processor and an ATWINC1500¹¹ WiFi module. The ATSAM21G18 is an ARM Cortex M0+ microcontroller with 32 KB of memory and a clock speed of 48 MHz. The development board is equipped with 20 GPIO pins, of which 10 are analog inputs. WiFi connectivity is added to this development board using the ATWINC1500 WiFi module. This module independently manages the WiFi connection of the device.

Power modes

The SAMD21 microcontroller has a power manager consisting of three parts: a clock controller, a reset controller and a sleep mode controller. The clock controller manages the power consumption of the peripherals embedded in the package by running clocks at a slower speed or disabling them. The reset controller governs resets of the microcontroller. Resetting the microcontroller can be done by different reset sources such as software resets and watchdog timers. The sleep mode controller manages the sleep modes of the device. There are two different sleep modes:

- **Idle mode** This is the lightest power state of this microcontroller and stops the CPU clock. Hence it is no longer possible to continue executing instructions on the CPU. Other clock sources are disabled based on the idle mode level. The other two clocks influenced by this sleep mode are the Advanced High-performance Bus (AHB) and Advanced Peripheral Bus (APB) clocks. Peripherals like timers and the USB controller are attached to these two clock sources.
- **Standby** This is the most power-efficient mode of this microcontroller. In this mode all clock sources are stopped, excluding clocks requested by peripherals or forced to run by the ONDEMAND and RUNSTDBY bits configuration. The SAMD21 can temporarily enable clock sources based on the requests of peripherals. This capability is called SleepWalking and configured with the ONDEMAND bit. Moreover, the regulator and RAM are put in low-power mode. The content of the RAM is preserved in this mode.

Item	Active	Idle 0	Idle 1	Idle 2	Standby
CPU clock	Running	Stopped	Stopped	Stopped	Stopped
AHB clock	Running	Running	Stopped	Stopped	Stopped
APB clock	Running	Running	Running	Stopped	Stopped
Oscillators	Running	Running	Running	Running	Running
		(if requested) ¹	(if requested) ¹	(if requested) ¹	(if requested) or stopped ²
Main clock	Running	Running	Running	Running	Stopped
Regulator mode	Normal	Normal	Normal	Normal	Low power
RAM mode	Normal	Normal	Normal	Normal	Low power
Power usage	3.7-4.8 mA	2.4 mA	1.8 mA	1.3 mA	3.4 μ A (RTC stopped), 4.6 μ A (RTC running)

¹ Depending on status of ONDEMAND bit.

² Depending on status of ONDEMAND and RUNSTDBY bit.

Table A.2: Overview of the sleep modes available on the ATSAM21G18. The precise conditions of the power measurements are mentioned in the datasheet (Data from Microchip¹²).

¹⁰See https://ww1.microchip.com/downloads/en/DeviceDoc/SAM_D21_DA1_Family_DataSheet_DS40001882F.pdf

¹¹See <https://ww1.microchip.com/downloads/en/DeviceDoc/70005304D.pdf>

The ATWINC1500 WiFi module is also equipped with a sleep mode. This mode can be automatically or manually enabled based on the power saving mode selected in the firmware¹³:

- **Manual** In this mode, the host device—i.e. the SAMD21 microcontroller—is responsible for enabling the sleep mode.
- **Deep automatic** This mode automatically manages the power states of the module. It works similar to the automatic sleep modes of the ESP8266. The device sleeps until an API function is called by the host¹⁴ or wakes up to receive the DTIM beacon. It is, on this module, also possible to save some extra energy by waking up every n th DTIM instead of every DTIM.

Device state	Current consumption
Transmitting	287 - 290 mA
Receiving	83 mA
Doze ¹	<390 μ A
Power down	<4 μ A

¹ Neither transmitting nor receiving

Table A.3: Power consumption of the ATWINC1500 in different device states. The precise conditions of the power measurements are mentioned in the datasheet (Data from Microchip¹⁵).

The power states of the WiFi module are in the mTask implementation automatically managed by the deep automatic power-saving mode. On top of that, the SAMD21 microcontroller uses the standby mode whenever possible. This sleep mode uses the least power but stops all clocks. Hence, the time returned by the `millis` function does not advance during sleep. This problem is solved by adding an offset to the time returned by the `millis` function. This approach does not work when the device wakes up by an interrupt. This problem is solved by using idle sleep mode when the device is waiting for interrupts. The `millis` timer advances in this mode, but it has a higher power consumption. Especially in the mTask implementation, it uses idle mode in a busy loop that waits for the said time as idle mode continuously wakes up by internally used interrupts. The lack of a good way to track time during standby mode is a point of improvement for future work as this solution leads to a time drift. Moreover, the WiFi connection is closed, and the WiFi module disabled if the sleep time is long enough.

Interrupts

An interrupt can be attached to all pins of the development board. It supports the same five interrupt types as the ESP8266. These five types are: high, low, falling, rising and changing.

¹²See https://ww1.microchip.com/downloads/en/DeviceDoc/SAM_D21_DA1_Family_DataSheet_DS40001882F.pdf

¹³See <https://ww1.microchip.com/downloads/en/DeviceDoc/ATWINC15x0-Wi-Fi-Network-Controller-Software-Design-Guide-DS00002389.pdf>

¹⁴The host's firmware wakes up the WiFi module when an API function is called

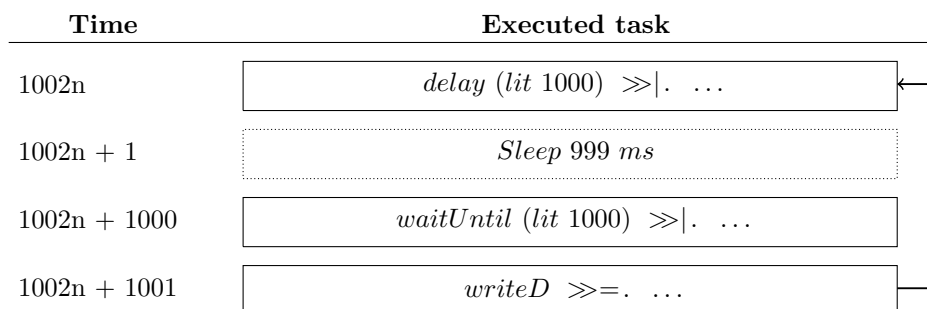
B Evaluation interval examples

B.1 Blink

Code:

```
blink :: Main (MTask v Bool) | mtask v
blink
= declarePin BuiltInLed PMOutput \led->
  fun \blink=(\x->
    delay (lit 1000)
    >>|. writeD led x
    >>=. blink o Not)
  In {main=blink (lit True)}
```

Timing diagram:



It is assumed that all evaluation steps take 1 ms.

Description:

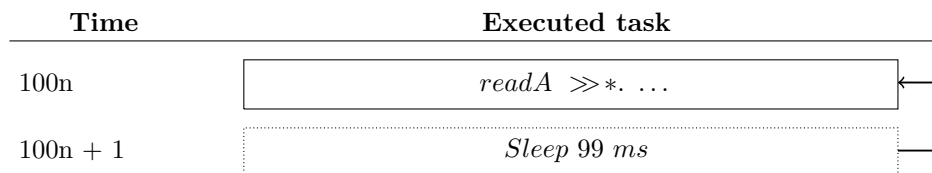
This blink task turns a LED on and off every second. Initially, the top node of the task tree is a $\gg|. .$ task with on the left a `delay` task and on the right a $\gg=.$ task. During the first evaluation, the `delay` task is translated into a `waitUntil` task. This `waitUntil` task emits an unstable value during its first evaluation. Consequently, the task tree stays unchanged, and the next evaluation contains no unevaluated nodes. The evaluation interval is thus equal to the refresh rate. The refresh rate is calculated as follows $\mathcal{R}(\text{waitUntil } 1000 \gg|. . (\text{writeD} \gg .. \dots)) \equiv \mathcal{R}(\text{waitUntil } 1000) \equiv [1000 - 0, 1000 - 0] \equiv [1000, 1000]$. The `waitUntil` task returns a stable value in the evaluation after the sleep time. Hence the step condition is met, and the rest of the task is evaluated. The evaluation interval is equal to $[0, 0]$ until the delay is reencountered.

B.2 Alarm

Code:

```
alarm :: Main (MTask v Bool) | mtask v
alarm
= declarePin A1 PMInput \a1->
  declarePin D4 PMOutput \d4->
  let alarm =
      readA a1
      >>*. [IfValue ((<=. ) (lit 10)) (\_.writeD d4 (lit True))]
  in {main = alarm}
```

Timing diagram:



It is assumed that all evaluation steps take 1ms.

Description:

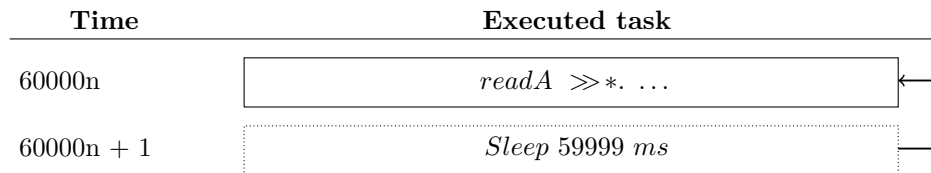
This example describes a task to monitor the value of an analog pin. Pin D4 is set to high when the value of the analog pin exceeds a specified threshold. The value of the analog pin is read at the frequency of the following refresh rate $\mathcal{R}(readA \gg *. [\dots]) \equiv \mathcal{R}(readA) \equiv [0, 100]$. The refresh rate of `readA` is equal to $[0, 100]$, because that is the default refresh rate of this task.

B.3 Alarm with custom refresh rate

Code:

```
alarm :: Main (MTask v Bool) | mtask v
alarm
= declarePin A1 PMInput \a1->
  declarePin D4 PMOutput \d4->
    let alarm =
        readA` (RangeSec (lit 30) (lit 60)) a1
        >>*. [IfValue ((<=.) (lit 10)) (\_.writeD d4 (lit True))]
    in {main = alarm}
```

Timing diagram:



It is assumed that all evaluation steps take 1 ms.

Description:

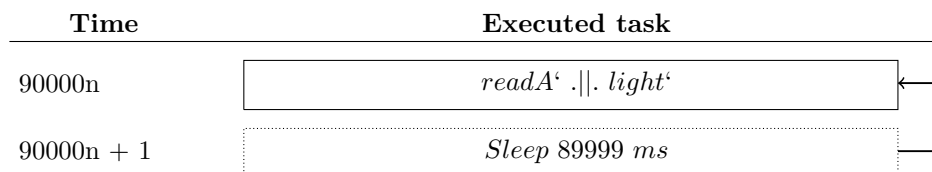
The third example is the same as the previous example but with a custom refresh rate assigned to the `readA` task. The refresh rate of the adjusted task is calculated as follows:
 $\mathcal{R}(\text{readA}' (\text{RangeSec } (\text{lit } 30) (\text{lit } 60))) \gg *. [\dots]) \equiv \text{readA}' (\text{RangeSec } (\text{lit } 30) (\text{lit } 60))) \equiv [30000, 60000]$

B.4 Plant monitor

Code:

```
monitorPlant :: Main (MTask v Bool) | mtask, dht, LightSensor v
monitorPlant
= declarePin MoistureSensorPin PMInput \m->
  lightsensor (i2c 0x23) \l->
    let moistureSensor =
        readA` (BeforeSec (lit 300)) m
        >>*. [IfValue ((<=.) (lit 100)) (\_. rtn (lit True))]
    in
    let lightSensor =
        light` (BeforeSec (lit 90)) l
        >>*. [IfValue ((<=.) (lit 150.0)) (\_. rtn (lit True))]
    in {main = lightSensor .||. moistureSensor}
```

Timing diagram:



It is assumed that all evaluation steps take 1 ms.

Description:

This example monitors the environment of a plant using a moisture and light sensor. The task emits a stable value if one of the values has a critical level. The refresh rate of the task is calculated as follows:

$$\begin{aligned}
 &\mathcal{R}(\\
 &\quad (readA' (BeforeSec (lit 300)) \gg*. [...]) .||. (light' (BeforeSec (lit 90)) \gg*. [...]) \\
 &) \equiv \mathcal{R}(\\
 &\quad ([0, 300000] \gg*. [...]) .||. ([0, 90000] \gg*. [...]) \\
 &) \equiv \mathcal{R}(\\
 &\quad [0, 300000] .||. [0, 90000] \\
 &) \equiv (\\
 &\quad [0, 300000] \cap_{safe} [0, 90000] \\
 &) \equiv [0, 90000]
 \end{aligned}$$

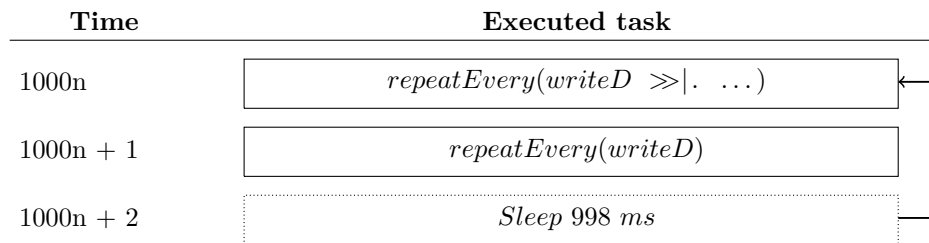
The resulting refresh rate corresponds to the expected refresh rate. The moisture level must be checked at least every 5 minutes and the light intensity at least every 90 seconds. Evaluating the task every 90 seconds is thus valid for all sensors.

B.5 Repeat with custom refresh rate

Code:

```
timedPulse :: Main (MTask v Bool) | mtask v
timedPulse = declarePin D0 PMOutput \d0->
  let pulse =
    writeD d0 (lit True)
    >>|. writeD d0 (lit False)
  in {main = rpeatEvery (ExactSec (lit 1)) pulse}
```

Timing diagram:



It is assumed that all evaluation steps take 1 ms.

Description:

This example pulses pin D0 every second shortly by setting pin D0 to high and back to low directly after. This process is repeatedly carried out using the `rpeatEvery` task. The `rpeatEvery` task reinitializes the child task if two conditions are met. The task is stable, and the time since the last reinitialization is larger than the lower bound of the user-defined refresh rate. The derived refresh rate of the `rpeatEvery` task is equal to $\mathcal{R}(\text{rpeatEvery } (\text{ExactSec } (\text{lit } 1)) t) \equiv [1000 - \text{startTime}, 1000 - \text{startTime}]$ after the loop body becomes stable.